Maxima by Example:
Ch. 1, Introduction to Maxima *

Edwin L. Woollett

August 11, 2009

Contents

1.1 What is **Maxima**? . . . 3
1.2 Which Maxima Interface Should You Use? . . . 4
1.3 Using the **wxMaxima** Interface . . . 4
1.3.1 Rational Simplification with **ratsimp** and **fullratsimp** . . . 9
1.4 Using the **Xmaxima** Interface . . . 11
1.5 Creating and Using a Startup File: **maxima-init.mac** . . . 16
1.6 Maxima Expressions, Numbers, Operators, Constants and Reserved Words . . . 18
1.7 Input and Output Examples . . . 20
1.8 Maxima Power Tools at Work . . . 21
1.8.1 The Functions **apropos** and **describe** . . . 21
1.8.2 The Function **ev** and the Properties **evflag** and **evfun** . . . 22
1.8.3 The List **functions** and the Function **fundef** . . . 24
1.8.4 The Function **kill** and the List **values** . . . 25
1.8.5 Examples of **map**, **fullmap**, **apply**, **grind**, and **args** . . . 25
1.8.6 Examples of **subst**, **ratsubst**, **part**, and **substpart** . . . 26
1.8.7 Examples of **coeff**, **ratcoef**, and **collectterms** . . . 28
1.8.8 Examples of **rat, diff, ratdiff, ratexpand, expand, factor, gfactor** and **partfrac** . . . 30
1.8.9 Examples of **integrate**, **assume**, **facts**, and **forget** . . . 33
1.8.10 Numerical Integration and Evaluation: **float, bfloat,** and **quad_qags** . . . 34
1.8.11 Taylor and Laurent Series Expansions with **taylor** . . . 35
1.8.12 Solving Equations: **solve, allroots, realroots,** and **find_root** . . . 37
1.8.13 Non-Rational Simplification: **radcan, logcontract, rootscontract,** and **radexpand** . . . 42
1.8.14 Trigonometric Simplification: **trigsimp, trigexpand, trigreduce,** and **trigrat** . . . 44
1.8.15 Complex Expressions: **rectform, demoivre, realpart, imagpart,** and **exponentialize** . . . 46
1.8.16 Are Two Expressions Numerically Equivalent? **zeroequiv** . . . 46
1.9 User Defined Maxima Functions: **define**, **fundef**, **block**, and **local** . . . 47
1.9.1 A Function Which Takes a Derivative . . . 47
1.9.2 Lambda Expressions . . . 50
1.9.3 Recursive Functions; **factorial**, and **trace** . . . 50
1.9.4 Non-Recursive Subscripted Functions (Hashed Arrays) . . . 51
1.9.5 Recursive Hashed Arrays and Memoizing . . . 52
1.9.6 Recursive Subscripted Maxima Functions . . . 53
1.9.7 Floating Point Numbers from a Maxima Function . . . 53
1.10 Pulling Out Overall Factors from an Expression . . . 55
1.11 Construction and Use of a Test Suite File . . . 56
1.12 History of Maxima's Development . . . 57

*This is a live document. This version uses **Maxima 5.19.0**. Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions for improvements to `woollett@charter.net`

Maxima by Example:
Ch. 2, Two Dimensional Plots and Least Squares Fits *

Edwin L. Woollett

September 16, 2010

Contents

2.1 Introduction to **plot2d** . . . 3
 2.1.1 First Steps with **plot2d** . . . 3
 2.1.2 **Parametric** Plots . . . 6
 2.1.3 Line Width and Color Controls . . . 9
 2.1.4 **Discrete** Data Plots: Point Size, Color, and Type Control . . . 12
 2.1.5 More **gnuplot_preamble** Options . . . 15
 2.1.6 Using **qplot** for Quick Plots of One or More Functions . . . 16
2.2 Least Squares Fit to Experimental Data . . . 18
 2.2.1 Maxima and Least Squares Fits: **lsquares_estimates** . . . 18
 2.2.2 Syntax of **lsquares_estimates** . . . 19
 2.2.3 Coffee Cooling Model . . . 20
 2.2.4 Experiment Data: **file_search, printfile, read_nested_list,** and **makelist** . . . 21
 2.2.5 Least Squares Fit of Coffee Cooling Data . . . 22

*This version uses **Maxima 5.18.1.** Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions to `woollett@charter.net`

Maxima by Example:
Ch. 3, Ordinary Differential Equation Tools *

Edwin L. Woollett

September 16, 2010

Contents

3.1 Solving Ordinary Differential Equations . . . 3
3.2 Solution of One First Order Ordinary Differential Equation (ODE) . . . 3
3.2.1 Summary Table . . . 3
3.2.2 Exact Solution with **ode2** and **ic1** . . . 3
3.2.3 Exact Solution Using **desolve** . . . 5
3.2.4 Numerical Solution and Plot with **plotdf** . . . 6
3.2.5 Numerical Solution with 4th Order Runge-Kutta: **rk** . . . 7
3.3 Solution of One Second Order ODE or Two First Order ODE's . . . 9
3.3.1 Summary Table . . . 9
3.3.2 Exact Solution with **ode2**, **ic2**, and **eliminate** . . . 9
3.3.3 Exact Solution with **desolve, atvalue,** and **eliminate** . . . 12
3.3.4 Numerical Solution and Plot with **plotdf** . . . 16
3.3.5 Numerical Solution with 4th Order Runge-Kutta: **rk** . . . 17
3.4 Examples of ODE Solutions . . . 19
3.4.1 Ex. 1: Fall in Gravity with Air Friction: Terminal Velocity . . . 19
3.4.2 Ex. 2: One Nonlinear First Order ODE . . . 22
3.4.3 Ex. 3: One First Order ODE Which is Not Linear in Y' . . . 23
3.4.4 Ex. 4: Linear Oscillator with Damping . . . 24
3.4.5 Ex. 5: Underdamped Linear Oscillator with Sinusoidal Driving Force . . . 28
3.4.6 Ex. 6: Regular and Chaotic Motion of a Driven Damped Planar Pendulum . . . 30
3.4.7 Free Oscillation Case . . . 31
3.4.8 Damped Oscillation Case . . . 32
3.4.9 Including a Sinusoidal Driving Torque . . . 33
3.4.10 Regular Motion Parameters Case . . . 33
3.4.11 Chaotic Motion Parameters Case. . . . 37
3.5 Using **contrib_ode** for ODE's . . . 43

*This version uses **Maxima 5.18.1** Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions to `woollett@charter.net`

Maxima by Example: Ch.4: Solving Equations *

Edwin L. Woollett

January 29, 2009

Contents

4 Solving Equations **3**
4.1 One Equation or Expression: Symbolic Solution or Roots 3
4.1.1 The Maxima Function solve 3
4.1.2 solve with Expressions or Functions & the multiplicities List 4
4.1.3 General Quadratic Equation or Function 5
4.1.4 Checking Solutions with subst or ev and a "Do Loop" 6
4.1.5 The One Argument Form of solve 7
4.1.6 Using disp, display, and print 7
4.1.7 Checking Solutions using map 8
4.1.8 Psuedo-PostFix Code: %% 9
4.1.9 Using an Expression Rather than a Function with Solve 9
4.1.10 Escape Speed from the Earth 11
4.1.11 Cubic Equation or Expression 14
4.1.12 Trigonometric Equation or Expression 14
4.1.13 Equation or Expression Containing Logarithmic Functions 15
4.2 One Equation Numerical Solutions: allroots, realroots, find_root 16
4.2.1 Comparison of realroots with allroots 17
4.2.2 Intersection Points of Two Polynomials 18
4.2.3 Transcendental Equations and Roots: find_root 21
4.2.4 find_root: Quote that Function! 23
4.2.5 newton 26
4.3 Two or More Equations: Symbolic and Numerical Solutions 28
4.3.1 Numerical or Symbolic Linear Equations with solve or linsolve 28
4.3.2 Matrix Methods for Linear Equation Sets: linsolve_by_lu 29
4.3.3 Symbolic Linear Equation Solutions: Matrix Methods 30
4.3.4 Multiple Solutions from Multiple Right Hand Sides 31
4.3.5 Three Linear Equation Example 32
4.3.6 Surpressing rat Messages: ratprint 34
4.3.7 Non-Linear Polynomial Equations 35
4.3.8 General Sets of Nonlinear Equations: eliminate, mnewton 37
4.3.9 Intersections of Two Circles: implicit_plot 37
4.3.10 Using Draw for Implicit Plots 38
4.3.11 Another Example 39
4.3.12 Error Messages and Do It Yourself Mnewton 42
4.3.13 Automated Code for mymnewton 45

*This version uses Maxima 5.17.1. This is a live document. Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions to `woollett@charter.net`

Maxima by Example: Ch.5: 2D Plots and Graphics using qdraw *

Edwin L. Woollett

January 29, 2009

Contents

5 2D Plots and Graphics using qdraw **3**
5.1 Quick Plots for Explicit Functions: ex(...) 3
5.2 Quick Plots for Implicit Functions: imp(...) 10
5.3 Contour Plots with contour(...) 12
5.4 Density Plots with qdensity(...) 14
5.5 Explicit Plots with Greater Control: ex1(...) 17
5.6 Explicit Plots with ex1(...) and Log Scaled Axes 19
5.7 Data Plots with Error Bars: pts(...) and errorbars(...) 21
5.8 Implicit Plots with Greater Control: imp1(...) 27
5.9 Parametric Plots with para(...) 29
5.10 Polar Plots with polar(...) 31
5.11 Geometric Figures: line(...) 32
5.12 Geometric Figures: rect(...) 34
5.13 Geometric Figures: poly(...) 35
5.14 Geometric Figures: circle(...) and ellipse(...) 38
5.15 Geometric Figures: vector(..) 40
5.16 Geometric Figures: arrowhead(..) 43
5.17 Labels with Greek Letters 43
5.17.1 Enhanced Postscript Methods 43
5.17.2 Windows Fonts Methods with jpeg Files 47
5.17.3 Using Windows Fonts with the Gnuplot Console Window 48
5.18 Even More with more(...) 49
5.19 Programming Homework Exercises 50
5.19.1 General Comments 50
5.19.2 XMaxima Tips 51
5.19.3 Suggested Projects 51
5.20 Acknowledgements 52

*This version uses Maxima 5.17.1. This is a live document. Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions to `woollett@charter.net`

Maxima by Example: Ch.6: Differential Calculus *

Edwin L. Woollett

October 21, 2010

Contents

6 Differential Calculus **3**

6.1 Differentiation of Explicit Functions . . . 4

6.1.1 All About **diff** . . . 4

6.1.2 The Total Differential . . . 5

6.1.3 Controlling the Form of a Derivative with **gradef** . . . 6

6.2 Critical and Inflection Points of a Curve Defined by an Explicit Function . . . 7

6.2.1 Example 1: A Polynomial . . . 7

6.2.2 Automating Derivative Plots with **plotderiv** . . . 9

6.2.3 Example 2: Another Polynomial . . . 11

6.2.4 Example 3: $\mathbf{x^{2/3}}$, Singular Derivative, Use of **limit** . . . 12

6.3 Tangent and Normal of a Point of a Curve Defined by an Explicit Function . . . 13

6.3.1 Example 1: $\mathbf{x^2}$ . . . 14

6.3.2 Example 2: $\mathbf{\ln(x)}$ . . . 14

6.4 Maxima and Minima of a Function of Two Variables . . . 15

6.4.1 Example 1: Minimize the Area of a Rectangular Box of Fixed Volume . . . 16

6.4.2 Example 2: Maximize the Cross Sectional Area of a Trough . . . 18

6.5 Tangent and Normal of a Point of a Curve Defined by an Implicit Function . . . 19

6.5.1 Tangent of a Point of a Curve Defined by $\mathbf{f(x,y) = 0}$ . . . 21

6.5.2 Example 1: Tangent and Normal of a Point of a Circle . . . 23

6.5.3 Example 2: Tangent and Normal of a Point of the Curve $\mathbf{\sin(2\,x)\ \cos(y) = 0.5}$ . . . 25

6.5.4 Example 3: Tangent and Normal of a Point on a Parametric Curve: $\mathbf{x = \sin(t), y = \sin(2\,t)}$ . . . 26

6.5.5 Example 4: Tangent and Normal of a Point on a Polar Plot: $\mathbf{x = r(t)\cos(t), y = r(t)\sin(t)}$ . . . 27

6.6 Limit Examples Using Maxima's **limit** Function . . . 28

6.6.1 Discontinuous Functions . . . 29

6.6.2 Indefinite Limits . . . 32

6.7 Taylor Series Expansions using **taylor** . . . 34

6.8 Vector Calculus Calculations and Derivations using **vcalc.mac** . . . 36

6.9 Maxima Derivation of Vector Calculus Formulas in **Cylindrical Coordinates** . . . 40

6.9.1 The Calculus Chain Rule in Maxima . . . 41

6.9.2 Laplacian $\nabla^2 f(\rho, \varphi, z)$ . . . 43

6.9.3 Gradient $\nabla f(\rho, \varphi, z)$ . . . 45

6.9.4 Divergence $\nabla \cdot \mathbf{B}(\rho, \varphi, z)$ . . . 48

6.9.5 Curl $\nabla \times \mathbf{B}(\rho, \varphi, z)$ . . . 49

6.10 Maxima Derivation of Vector Calculus Formulas in **Spherical Polar Coordinates** . . . 50

*This version uses Maxima 5.21.1 This is a live document. Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions to `woollett@charter.net`

Maxima by Example:
Ch.7: Symbolic Integration *

Edwin L. Woollett

September 16, 2010

Contents

7.1 Symbolic Integration with **integrate** 3
7.2 Integration Examples and also **defint**, **ldefint**, **beta**, **gamma**, **erf**, and **logabs** 4
7.3 Piecewise Defined Functions and **integrate** 12
7.4 Area Between Curves Examples 14
7.5 Arc Length of an Ellipse 17
7.6 Double Integrals and the Area of an Ellipse 19
7.7 Triple Integrals: Volume and Moment of Inertia of a Solid Ellipsoid 22
7.8 Derivative of a Definite Integral with Respect to a Parameter 24
7.9 Integration by Parts 28
7.10 Change of Variable and **changevar** 29

*This version uses **Maxima 5.18.1**. This is a live document. Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions to `woollett@charter.net`

Maxima by Example:
Ch.8: Numerical Integration *

Edwin L. Woollett

September 16, 2010

Contents

8.1 Introduction 3
8.2 Numerical Integration Basic Tools: **quad_qags**, **romberg**, **quad_qagi** 3
8.2.1 Syntax for Quadpack Functions 3
8.2.2 Ouput List of Quadpack Functions and Error Code Values 3
8.2.3 Integration Rule Parameters and Optional Arguments 4
8.2.4 **quad_qags** for a Finite Interval 4
8.2.5 **romberg** for a Finite Interval 7
8.2.6 **quad_qags** and **romberg** for Numerical Double Integrals 8
8.2.7 **quad_qagi** for an Infinite or Semi-infinite Interval 10
8.3 Numerical Integration: Sharper Tools 12
8.3.1 **quad_qag** for a General Oscillatory Integrand 12
8.3.2 **quad_qawo** for Fourier Series Coefficients 14
8.3.3 **quad_qaws** for End Point Algebraic and Logarithmic Singularities 15
8.3.4 **quad_qawc** for a Cauchy Principal Value Integral 17
8.3.5 **quad_qawf** for a Semi-Infinite Range Cosine or Sine Fourier Transform 19
8.4 Finite Region of Integration Decision Tree 20
8.5 Semi-infinite or Infinite Region of Integration Decision Tree 21

*This version uses **Maxima 5.18.1**. This is a live document. Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions to `woollett@charter.net`

Maxima by Example:
Ch.9: Bigfloats and Arbitrary Precision Quadrature *

Edwin L. Woollett

September 16, 2010

Contents

9.1 Introduction 4
9.2 The Use of Bigfloat Numbers in Maxima 4
9.2.1 Bigfloat Numbers Using **bfloat**, **fpprec**, and **fpprintprec** 4
9.2.2 Using **print** and **printf** with Bigfloats 8
9.2.3 Adding Bigfloats Having Differing Precision 11
9.2.4 Polynomial Roots Using **bfallroots** 12
9.2.5 Bigfloat Number Gaps and Binary Arithmetic 14
9.2.6 Effect of Floating Point Precision on Function Evaluation 15
9.3 Arbitrary Precision Quadrature with Maxima 16
9.3.1 Using **bromberg** for Arbitrary Precision Quadrature 16
9.3.2 A **Double Exponential** Quadrature Method for $\mathbf{a} \leq \mathbf{x} < \infty$ 19
9.3.3 The **tanh-sinh** Quadrature Method for $\mathbf{a} \leq \mathbf{x} \leq \mathbf{b}$ 22
9.3.4 The **Gauss-Legendre** Quadrature Method for $\mathbf{a} \leq \mathbf{x} \leq \mathbf{b}$ 28

*This version uses **Maxima 5.18.1**. This is a live document. Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions to `woollett@charter.net`

Maxima by Example: Ch.10: Fourier Series, Fourier and Laplace Transforms *

Edwin L. Woollett

September 16, 2010

Contents

10.1 Introduction 3
10.2 Fourier Series Expansion of a Function 3
10.2.1 Fourier Series Expansion of a Function over $(-\pi, \pi)$ 3
10.2.2 Fourier Series Expansion of $f(x) = x$ over $(-\pi, \pi)$ 4
10.2.3 The **calculus/fourie.mac** Package: **fourier**, **foursimp**, **fourexpand** 5
10.2.4 Fourier Series Expansion of a Function Over $(-p, p)$ 7
10.2.5 Fourier Series Expansion of the Function $|x|$ 8
10.2.6 Fourier Series Expansion of a Rectangular Pulse 11
10.2.7 Fourier Series Expansion of a Two Element Pulse 13
10.2.8 Exponential Form of a Fourier Series Expansion 16
10.3 Fourier Integral Transform Pairs 18
10.3.1 Fourier Cosine Integrals and **fourintcos(..)** 18
10.3.2 Fourier Sine Integrals and **fourintsin(..)** 19
10.3.3 Exponential Fourier Integrals and **fourint** 21
10.3.4 Example 1: Even Function 21
10.3.5 Example 2: Odd Function 24
10.3.6 Example 3: A Function Which is Neither Even nor Odd 26
10.3.7 Dirac Delta Function $\delta(x)$ 28
10.3.8 Laplace Transform of the Delta Function Using a Limit Method 31
10.4 Laplace Transform Integrals 32
10.4.1 Laplace Transform Integrals: **laplace(..)**, **specint(..)** 32
10.4.2 Comparison of **laplace** and **specint** 32
10.4.3 Use of the Dirac Delta Function (Unit Impulse Function) **delta** with **laplace(..)** 36
10.5 The Inverse Laplace Transform and Residues at Poles 36
10.5.1 **ilt**: Inverse Laplace Transform 36
10.5.2 **residue**: Residues at Poles 37

---

*This version uses **Maxima 5.18.1**. This is a live document. Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions to `woollett@charter.net`

Maxima by Example:
Ch.11: Fast Fourier Transform Tools *

Edwin L. Woollett

August 13, 2009

Contents

11.1 Examples of the Use of the Fast Fourier Transform Functions **fft** and **inverse_fft** 3

11.1.1 Example 1: FFT Spectrum of a Monochromatic Signal 3

11.1.2 Example 2: FFT Spectrum of a Sum of Two Monochromatic Signals 8

11.1.3 Example 3: FFT Spectrum of a Rectangular Wave 10

11.1.4 Example 4: FFT Spectrum Sidebands of a Tone Burst Before and After Filtering 13

11.1.5 Example 5: Cleaning a Noisy Signal using FFT Methods 17

11.2 Our Notation for the Discrete Fourier Transform and its Inverse 22

11.3 Syntax of **qfft.mac** Functions 25

11.4 The Discrete Fourier Transform Derived via a Numerical Integral Approximation 27

11.5 Fast Fourier Transform References 28

*This is a live document. This version uses **Maxima 5.19.0**. Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions for improvements to `woollett@charter.net`

Maxima by Example:
Ch. 12, Dirac Algebra and Quantum Electrodynamics *

Edwin L. Woollett

September 13, 2010

Contents

12.1 References . . . 3
12.2 High Energy Physics Notation Conventions . . . 3
12.3 Introduction to the Dirac Package . . . 6
12.4 traceConEx.mac: Examples of Frequently Used Functions in the Dirac Package . . . 7
12.4.1 Dirac Spinors . . . 12
12.5 moller0.mac: Scattering of Identical Scalar Charged Particles . . . 16
12.6 moller1.mac: High Energy Elastic Scattering of Two Electrons . . . 20
12.7 moller2.mac: Arbitrary Energy Moller Scattering . . . 26
12.8 bhabha1.mac: High Energy Limit of Bhabha Scattering . . . 30
12.9 bhabha2.mac: Arbitrary Energy Bhabha Scattering . . . 33
12.10 photon1.mac: Photon Transverse Polarization 3-Vector Sums . . . 37
12.11 Covariant Physical External Photon Polarization 4-Vectors - A Review . . . 39
12.12 compton0.mac: Compton Scattering by a Spin 0 Structureless Charged Particle . . . 39
12.13 compton1.mac: Lepton-photon scattering . . . 43
12.14 pair1.mac: Unpolarized Two Photon Pair Annihilation . . . 48
12.15 pair2.mac: Polarized Two Photon Pair Annihilation Amplitudes . . . 50
12.16 moller3.mac: Squared Polarized Amplitudes Using Symbolic or Explicit Matrix Trace Methods . . . 61
12.17 List of Dirac Package Files and Example Batch Files . . . 68

*This version uses **Maxima 5.21.1** Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions to `woollett@charter.net`

Maxima by Example:
Ch. 1, Introduction to Maxima *

Edwin L. Woollett

August 11, 2009

Contents

1.1 What is **Maxima**? . . . 3
1.2 Which Maxima Interface Should You Use? . . . 4
1.3 Using the **wxMaxima** Interface . . . 4
1.3.1 Rational Simplification with **ratsimp** and **fullratsimp** . . . 9
1.4 Using the **Xmaxima** Interface . . . 11
1.5 Creating and Using a Startup File: **maxima-init.mac** . . . 16
1.6 Maxima Expressions, Numbers, Operators, Constants and Reserved Words . . . 18
1.7 Input and Output Examples . . . 20
1.8 Maxima Power Tools at Work . . . 21
1.8.1 The Functions **apropos** and **describe** . . . 21
1.8.2 The Function **ev** and the Properties **evflag** and **evfun** . . . 22
1.8.3 The List **functions** and the Function **fundef** . . . 24
1.8.4 The Function **kill** and the List **values** . . . 25
1.8.5 Examples of **map**, **fullmap**, **apply**, **grind**, and **args** . . . 25
1.8.6 Examples of **subst**, **ratsubst**, **part**, and **substpart** . . . 26
1.8.7 Examples of **coeff**, **ratcoef**, and **collectterms** . . . 28
1.8.8 Examples of **rat, diff, ratdiff, ratexpand, expand, factor, gfactor** and **partfrac** . . . 30
1.8.9 Examples of **integrate**, **assume**, **facts**, and **forget** . . . 33
1.8.10 Numerical Integration and Evaluation: **float, bfloat,** and **quad_qags** . . . 34
1.8.11 Taylor and Laurent Series Expansions with **taylor** . . . 35
1.8.12 Solving Equations: **solve, allroots, realroots,** and **find_root** . . . 37
1.8.13 Non-Rational Simplification: **radcan, logcontract, rootscontract,** and **radexpand** . . . 42
1.8.14 Trigonometric Simplification: **trigsimp, trigexpand, trigreduce,** and **trigrat** . . . 44
1.8.15 Complex Expressions: **rectform, demoivre, realpart, imagpart,** and **exponentialize** . . . 46
1.8.16 Are Two Expressions Numerically Equivalent? **zeroequiv** . . . 46
1.9 User Defined Maxima Functions: **define**, **fundef**, **block**, and **local** . . . 47
1.9.1 A Function Which Takes a Derivative . . . 47
1.9.2 Lambda Expressions . . . 50
1.9.3 Recursive Functions; **factorial**, and **trace** . . . 50
1.9.4 Non-Recursive Subscripted Functions (Hashed Arrays) . . . 51
1.9.5 Recursive Hashed Arrays and Memoizing . . . 52
1.9.6 Recursive Subscripted Maxima Functions . . . 53
1.9.7 Floating Point Numbers from a Maxima Function . . . 53
1.10 Pulling Out Overall Factors from an Expression . . . 55
1.11 Construction and Use of a Test Suite File . . . 56
1.12 History of Maxima's Development . . . 57

*This is a live document. This version uses **Maxima 5.19.0**. Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions for improvements to `woollett@charter.net`

```
COPYING AND DISTRIBUTION POLICY
This document is part of a series of notes titled
"Maxima by Example" and is made available
via the author's webpage http://www.csulb.edu/~woollett/
to aid new users of the Maxima computer algebra system.

NON-PROFIT PRINTING AND DISTRIBUTION IS PERMITTED.

You may make copies of this document and distribute them
to others as long as you charge no more than the costs of printing.

These notes (with some modifications) will be published in book form
eventually via Lulu.com in an arrangement which will continue
to allow unlimited free download of the pdf files as well as the option
of ordering a low cost paperbound version of these notes.
```


Feedback from readers is the best way for this series of notes to become more helpful to new users of Maxima. *All* comments and suggestions for improvements will be appreciated and carefully considered.

```
LOADING FILES
The defaults allow you to use the brief version load(fft) to load in the
Maxima file fft.lisp.
To load in your own file, such as qxxx.mac
using the brief version load(qxxx), you either need to place
qxxx.mac in one of the folders Maxima searches by default, or
else put a line like:

file_search_maxima : append(["c:/work2/###.{mac,mc}"],file_search_maxima )$

in your personal startup file maxima-init.mac (see later in this chapter
for more information about this).

Otherwise you need to provide a complete path in double quotes,
as in load("c:/work2/qxxx.mac"),

We always use the brief load version in our examples, which are generated
using the Xmaxima graphics interface on a Windows XP computer, and copied
into a fancy verbatim environment in a latex file which uses the fancyvrb
and color packages.
```

```
  Maxima, a Computer Algebra System.
  Some numerical results depend on the Lisp version used.
  This chapter uses Version 5.19.0 (2009) using Lisp GNU
  Common Lisp (GCL) GCL 2.6.8 (aka GCL).
  http://maxima.sourceforge.net/
```

Acknowledgements

Some of the examples used in these notes are from the Maxima html help manual or the Maxima mailing list:
`http://maxima.sourceforge.net/maximalist.html`.

Our discussion begins with some of the "nuts and bolts" of using the software in a Windows XP environment, and continues with information useful to a new user.

The author would like to thank the Maxima developers for their friendly help via the Maxima mailing list.

1.1 What is Maxima?

Maxima is a powerful computer algebra system (CAS) which combines symbolic, numerical, and graphical abilities. See the Maxima sourceforge webpage `http://maxima.sourceforge.net/`.

A cousin of the commercial Macsyma CAS (now available but without support), Maxima is a freely available and open source program which is being continuously improved by a team of volunteers. When compared with Mathematica or Maple, Maxima has a more basic interface, but has the advantage in price (!). Students, teachers, and researchers can "own" multiple copies for home, laptop, and desktop without the expense of buying licenses for each copy.

There are known "bugs" in the present version (a new version is available about three times each year), and the volunteer developers and programming experts are dealing with these known bugs as time permits.

Maxima is not only "free" and will always stay that way, but also comes with a copy of the underlying source code (in a dialect of the Lisp language), which a user can modify to suit her own research needs and then share with the Maxima community of users.

A self-installing binary for Windows users is available, making it easy for Windows users to get a fast start.

Here is a more technical description of Maxima, taken from the Unix/Linus man document:

> Maxima is a version of the MIT-developed MACSYMA system, modified to run under CLISP. It is an interactive expert system and programming environment for symbolic and numerical mathematical manipulation. Written in Lisp, it allows differentiation, integration, solution of linear or polynomial equations, factoring of polynomials, expansion of functions in Laurent or Taylor series, computation of Poisson series, matrix and tensor manipulations, and two- and three-dimensional graphics.
>
> Procedures may be written using an ALGOL-like syntax, and both Lisp-like functions and pattern matching facilities are provided. Files containing Maxima objects may be read from and written to disk files. Pre-written Maxima commands may be read from a file and executed, allowing batch-mode use.
>
> Maxima is a complex system. It includes both known and unknown bugs. Use at your own risk. The Maxima bug database is available at
> `http://sourceforge.net/tracker/?atid=104933&group_id=4933&func=browse`.
>
> New bug reports are always appreciated. Please include the output of the Maxima function **build_info()** with the report.

Information about the history of Maxima and its relation to Macsyma can be found in the last section of this chapter.

You should first familiarize yourself with the Maxima work environment by downloading and installing Maxima on your computer, and starting up either the wxMaxima or the XMaxima interface.

1.2 Which Maxima Interface Should You Use?

New users generally like to start with **wxMaxima** since there are convenient icons which help locate Maxima functions for common tasks. **wxMaxima** allows the user to construct a combination text, calculation, and plot document which can be saved and used as a homework submission, used as a tutorial for others and/or simply used as a "permanent" record of work on some topic. For new users, the menus and buttons allow a gradual learning of Maxima syntax, by reading what the menus and buttons do with an expression, and the user can pick up a knowledge of most of the "power tools" in Maxima in this way. The reader should be warned, however, that no menu and button system can include every Maxima function which might be either useful or needed for a particular task.

Experienced users tend to split between **wxMaxima** and **Xmaxima**, switching to **Xmaxima** because they already know the names of common Maxima functions which help in getting the job done, and appreciate a simple stable interface without distractions. Experienced users tend to become good "touch typists", able to type most things without looking at the keyboard and using both hands. Such experienced users usually find that it is faster to just type the name than to reach for the mouse and manipulate the cursor to the right button. The **Xmaxima** interface is quite stable between new versions of Maxima, whereas the **wxMaxima** interface is being actively developed and changes to the appearance and behavior occur frequently during this period of rapid development.

Xmaxima is also a faster environment for testing and playing with code ideas, and the session record can be easily copied and pasted into a Latex verbatim environment with zero hassle. The current version of **wxMaxima** does not provide this hassle free transfer to a latex document (although one can save output as an image, but then one must go through the hassle of converting to eps file image format if one is using the conventional latex to dvi to pdf route).

1.3 Using the wxMaxima Interface

From the `wxMaxima` webpage `http://wxmaxima.sourceforge.net/wiki/index.php/Main_Page` which could be accessed via

```
start, My Programs, Maxima-5.19.0, wxMaxima on the Web
```

one finds the information:

```
wxMaxima features include:
  * 2D formatted math display: wxMaxima implements its
          own math display engine to nicely display maxima output.
  * Menu system: most Maxima commands are available through menus.
      Most used functions are also available through a button
          panel below the document.
  * Dialogs: commands which require more that one argument can
          be entered through dialogs so that there is no need to
          remember the exact syntax.
  * Create documents: text can be mixed with math
      calculations to create documents. Documents can be
          saved and edited again later.
  * Animations: version 0.7.4 adds support for simple animations.
```

On that page is a link to a set of `wxMaxima` tutorials on the page
`http://wxmaxima.sourceforge.net/wiki/index.php/Tutorials`.

There you can download tutorials in the form of zip files, which, when unzipped, become wxMaxima document format (wxm) files which can then be loaded into your wxMaxima work session.

`10minute.zip` contains a "ten minute" wxMaxima tutorial. `usingwxmaxima.zip` provides general information about the cell structure of wxMaxima.

Starting wxMaxima

One can use the Windows Start menu:

```
start, All Programs, Maxima-5.19.0, wxMaxima
```

During the Windows binary setup process you can select the options which place icons for both **wxMaxima** and **XMaxima** on your desktop for a convenient start, and you can later copy any shortcut and paste it into your work folder for an alternative start method.

Quitting wxMaxima

The quick way to quit is the two-key command **Ctrl + q**.

The Maxima Manual

To access the Maxima manual from within **wxMaxima**, you can use function key **F1** or the menu item
Help, Maxima Help.

The Online wxMaxima Forum

You can access the **wxMaxima** online forum at the web page

```
http://sourceforge.net/forum/forum.php?forum_id=435775
```

which can be accessed using

```
Start, My Programs, Maxima-5.19.0, wxMaxima Online Forum
```

and search for topics of interest.

The Cell Structure of wxMaxima

In the second tutorial, Using wxMaxima, the cell structure of a wxMaxima document is explained (we will use parts of that discussion here and we give only a rough idea of the actual appearance here):

The top of the cell bracket is actually a triangle. The following is a “text cell” which has no Maxima code.

```
-----
|
| Unlike "command-line Maxima"' (such as XMaxima or Console mode)
| which works in a simple input-output manner, wxMaxima introduces
| the concept of a live mathematical document, in which you mix
| text, calculations and plots.
|
| Each wxMaxima document consists of a number of so called "cells".
| The cell is the basic building block of a wxMaxima document. Each cell has a
| bracket on the left border of the document, indicating where the cell
| begins and ends. Cells are of different types. You can have a "title
| cell", a "section cell", and a "text cell" like this one. The most
| important cell type is the "input cell".
---
```

The next cell is an example of an input cell, which has the input prompt **>>** at the top. Note that we can include a text comment within an input cell (which will be ignored by the Maxima computational engine) by putting the comment between the delimiters **/*** and ***/** :

```
----
| >> /* this is an input cell - it holds Maxima code and can be
| evaluated by first left-clicking once anywhere in the cell and
| then using the two-key command SHIFT-ENTER. The code entered in this cell
| will be sent to Maxima when you press SHIFT-ENTER. Before
| wxMaxima sends code to Maxima, it checks if the contents
| of each code command in this input cell ends with a ';' or a '$'.
| If it doesn't, wxMaxima adds a ';' at the end. Maxima requires that
| each completed Maxima statement end with either ';' or '$'.
| Note that this does not mean you have to have each statement on
| one line. An input cell could have the contents
|     sin
|     (
|      x
|       );
|
| and this would be accepted as a complete Maxima input, equivalent to
|    sin(x);
|
| Any *output* wxMaxima gets from Maxima will be attached to the end of
| the input cell. Try clicking in this cell and pressing SHIFT-ENTER. */
|
|   /*  example Maxmima code: */
|
|   print("Hello, world!")$
|   integrate(x^2, x);
---
```

If you click once somewhere inside this cell the vertical left line changes to (if you have not changed the default) bright red, indicating that the cell has been selected for some action. If you then press the two-key combination **Shift + Enter**, the prompt **>>** will be replaced by **(%i1)**, and the results of the two separate Maxima commands will be shown at the bottom of the cell as:

```
|  Hello, world!
|             3
|            x
|    (%o2)  ---
|            3
----
```

except that the results are shown using pixel graphics with much better looking results than we show here.

There is no **(%o1)** since the **print** command was followed by the **$** sign, which surpresses the normal default output.

If you have either a blank screen or a horizontal line present and just start typing, an input type cell is automatically created. (You can instead, prior to typing, use the function key **F7** to create a new input cell and can use the function key **F6** to create a new text cell).

In the current version of wxMaxima, if you change windows to some other task unrelated to wxMaxima, a junk input cell may be created in wxMaxima containing something which looks like a percent sign **>> %** which you will see when you come back to your wxMaxima window. If you want to have an input cell at that location, just backspace to get rid of the weird symbol and continue.

If you then want to delete this junk cell, (or any cell you might be working on) just left-click on the bottom of the cell bracket (which will highlight the bracket) and press the **Delete** key (or else select **Edit, Cell, Cut Cell**).

An alternative method is to use the horizontal line as a vertical position cursor which acts on cells. If your cursor is still inside a cell you want to delete, use the DOWN key to get the cursor out of the cell, and the horizontal black line appears. At that point you can press the backspace (or delete) key twice to delete the cell above the black horizontal line. Or you can press SHIFT-UP once to select that cell and then press DELETE. Using SHIFT-UP more than once will select a group of cells for action.

In the author's copy of **wxMaxima** (ie., using Windows XP), trying to use the usual Windows menu key combination **Alt + E**, for example, does not actuate the **wxMaxima** Edit menu icon; one must left-click on that Edit icon at the top of the screen to choose among the options.

The next cell is a "text cell" which does not need Maxima comment delimiters `/*` and `*/`.

```
----
|
| Again, there is a triangle at the top of both text and
|  input type cells which we don't show. An open triangle is the
|  default, but if you click on the triangle, it will turn solid
|  black and a) for a text cell, the text content is hidden, and
|  b) for an input type cell, the Maxima ouput of the cell is hidden.
|  Clicking that solid black triangle again will restore the hidden
|  portions to view.
|
|  Editing cells is easy. To modify the contents of a cell,
|  click into it (ie., left-click anywhere in the cell once).
|  A cursor should appear and the left cell bracket should turn red,
|  (in default style mode) indicating that the cell is ready to be edited.
|
|  The usual Windows methods can be used to select parts of a cell.
|  One method is to hold the left mouse button down while you drag
|  over your selection. A second method combines the SHIFT key with
|  other keys to make the selection. For example, left-click at
|  a starting location. Then use the two-key command SHIFT-END to
|  select to the end of the line (say) or SHIFT-RIGHTARROW to
|  select part of the line. You can then use CTRL-C  to copy your
|  selection to the clipboard. You can then left-click somewhere
|  else and use CTRL-V to paste the clipboard contents at that location.
|  (This is the usual behvior: if you experience lack of correct pasting,
|  you can temporarily revert to the longer process of using the
|  Edit, Copy, and Edit, Paste menu items to get wxMaxima in the
|   right spirit).
|  The DOWN arrow key will step down one line at a time through the
|  cell lines and then make a horizontal line under the cell. You can
|  also simply left-click in the space between cells to create the
|  active horizontal line, which you can think of as a sort of
|  cursor for vertical position. With the horizontal line present,
|  simply start typing to automatically create a new input type cell.
|  Or press the function key F6 to create a new text type cell
|  at that vertical location.
--
```

When you're satisfied with the document you've created, save it using the two-key **Ctrl + s** command or the **File, Save** menu command. Note that the output parts of input cells will not be saved. When you load your document later, you can evaluate all cells by using the **Edit, Cell, Evaluate all cells** menu command or the shortcut two-key command **Ctrl + r**. If the shortcut two-key command doesn't work the first time, just repeat once and it should work. The present version of

wxMaxima is a little cranky still. This will evaluate all the cells in the order they were originally created.

Some common Maxima commands are available through the seven menu icons: **Maxima, Equations, Algebra, Calculus, Simplify, Plot and Numeric**. All of the menu choices which end with `.....` will open a dialog, to help you formulate your desired command. The resulting command will be inserted at the current horizontal cursor's position or below the currently active cell. The chosen command will also be evaluated.

Configuring the Buttons and Fonts

The bottom button panel can be configured through **Edit, Configure, Options Tab, Button Panel, Full or Basic**, and the font type and size can be configured through **Edit, Configure, Style Tab, Default Font, Choose Font, Times New Roman, Regular, 12** , for 12 pt generic roman for example. This is smaller than the startup default font, and may be too small for some users, but will allow more information to be seen per screen.

The default text cell background color is a six variable custom green that is nice. In the following, the italic box is left unchecked unless mentioned otherwise. To change a color, click the color bar once. The author combines the 12 pt roman choice with input labels in bold red, Maxima input in blue bold, text cell in bold black, strings in black bold italic, Maxima questions in blue bold, and all of the following in bold black: output labels, function names, variables, numbers, special constants, greek constants.

When you want to save your choices, select the Save button, which will write a file **style.ini** to the folder in which wx-Maxima was started.

(The author uses a shortcut to wxMaxima placed in his `c:\work2` windows xp folder, since that folder is where the author expects saved wxm and xml files to be saved. By starting with the contents of that folder in view, the author can then simply click on the wxMaxima shortcut link to begin work with wxMaxima in that folder.)

The author chose the **Full** bottom button option (the default is **Basic**), which draws twenty buttons in two rows at the bottom of the screen. The top row contains the **Simplify, Simplify(r), Factor, Expand, Simplify(tr), Expand(tr), Reduce(tr), Rectform, Sum..., Product...** buttons.
The bottom row contains the **Solve..., Solve ODE..., Diff..., Integrate..., Limit..., Series..., Subst..., Map..., Plot2D..., Plot2D...** buttons.
Note that if you have made a selection before you left-click a button panel command or a menu command, that selection will be used as the main (or only) argument to the command. Selection of a previous output or part of an input will work in a similar manner. The selected button function will act on the input cell contents or selection and immediately proceed with evaluation unless there is an argument dialog which must be completed.

Multiple Maxima Requests in One Input Cell

Here is a calculation of the cross sectional area `a` and volume `v` of a right circular cylinder in terms of the cross section radius `r` and the height `h` (ignoring units). We first create an input cell as described above, by just starting to type the following lines, ending each line with the ENTER key to allow entry of the following line.

```
|---
| >>  (r : 10, h : 100)$
|      a : %pi * r^2;
|      v : a * h;
---
```

Note that we put two Maxima assignment statements on one line, using the syntax **(job_1, job_2)$**. Naturally, you can put more than two statements between the beginning and ending parentheses. Thus far, the Maxima computational engine has not been invoked. The left cell bracket will be bright red (in the default style), and you will have a blinking cursor in the cell somewhere.

If you now use the two-key command SHIFT-ENTER, all Maxima commands will be carried out and the ouputs will show up (in the same order as the inputs) at the bottom of that input cell, looking like

```
----
| (%i1)  (r : 10, h : 100)$
|         a : %pi * r^2;
|         v : a * h;
| (%o2)   100*%pi
| (%o3)   10000*%pi
---
```

with the horizontal (vertical location) line present underneath. The input prompt **>>** was replaced with the input number **(%i1)**. Output **(%o1)** was not printed to the screen because the first input ended with **$**.

Now just start typing to start the next input cell:

```
----
| >>   [ a, v ], numer ;
---
```

With the cell bracket highlighted in red and the blinking cursor inside the cell, press SHIFT-ENTER to carry out the operation, which returns the numerical value of the cross-sectional area **a** and the cylinder volume **v** as a list.

```
----
| (%i4)    [ a, v ], numer ;
| (%o4)   [314.1592653589793,31415.92653589793]
```

1.3.1 Rational Simplification with ratsimp and fullratsimp

A rational expression is a ratio of polynomials or a sum of such. A particular case of a rational expression is a polynomial (whether expanded out or factored). **ratsimp** and **fullratsimp** can be useful tools for expressions, part of whose structure is of this form. Here is a simple example. If you are using wxMaxima, just start typing the expression you see entered below:

```
----
| >>  (x+2)*(x-2)
---
```

When you type a leading parenthesis **'('**, the trailing parenthesis **')'** also appears automatically. However, if you type an ending parenthesis anyway, wxMaxima will not use it, but rather will properly end with one parenthesis. However, you should pay attention to this feature at first so you understand how it works. Un-needed parentheses will result in a Maxima error message.

Now press HOME and then SHIFT-END to select the whole expression, **(x+2)*(x-2)**, then click on the **Simplify** button at the bottom of the screen. Maxima will apply a default type of simplification using the function **ratsimp**, and the cell contents evaluation will show

```
|(%i5) ratsimp((x+2)*(x-2));
|        2
|(%o5)  x  - 4
```

Thus the **Simplify** button uses **ratsimp** to simplify an expression. Instead of using the **Simplify** button, you could use the menu selection: **Simplify, Simplify Expression**. The word "simplify" is being used here in a loose fashion, since Maxima has no way of knowing what a particular user will consider a useful simplification.

A nice feature of wxMaxima is that you can left-click on a Maxima function name (in an input cell) like **ratsimp** and then press the function key **F1**, and the Maxima manual will open up to the entry for that function.

Note that if you had started with the input cell contents

```
----
| >> log( (x+2)*(x-2) ) + log(x)
---
```

and then highlighted just the **(x+2)*(x-2)** portion, followed by clicking on the **Simplify** button, you would get the same input and output we see above. Thus Maxima will ignore the rest of the expression. On the other hand, if you highlighted the **whole expression** and then clicked on **Simplify**, you would get

```
----
| (%i5)    ratsimp(log((x+2)*(x-2))+log(x));
|                                 2
| (%o5)                     log(x  - 4) + log(x)
---
```

fullratsimp

According to the Maxima manual

> **fullratsimp** repeatedly applies **ratsimp** followed by non-rational simplification to an expression until no further change occurs, and returns the result. When non-rational expressions are involved, one call to **ratsimp** followed as is usual by non-rational (”general”) simplification may not be sufficient to return a simplified result. Sometimes, more than one such call may be necessary. **fullratsimp** makes this process convenient.

Here is the Maxima manual example for **fullratsimp**, making use of the **Xmaxima** interface.

```
(%i1) expr: (x^(a/2) + 1)^2*(x^(a/2) - 1)^2/(x^a - 1);

                           a/2     2   a/2     2
                         (x    - 1)  (x    + 1)
(%o1)                    -----------------------
                                  a
                                 x  - 1
(%i2) expr, ratsimp;
                             2 a      a
                            x    - 2 x  + 1
(%o2)                       ---------------
                                  a
                                 x  - 1
(%i3) expr, fullratsimp;
                                  a
(%o3)                            x  - 1
(%i4) rat (expr);

                           a/2 4       a/2 2
                         (x   )  - 2 (x   )  + 1
(%o4)/R/                 -----------------------
                                  a
                                 x  - 1
```

We see that **fullratsimp** returned a completely simplified result. **rat(expr)** converts **expr** to a **canonical rational expression** (CRE) form by expanding and combining all terms over a common denominator and cancelling out the greatest common divisor of the numerator and denominator, as well as converting floating point numbers to rational numbers within a tolerance of **ratepsilon** (see the Manual for more information about **rat**).

1.4 Using the Xmaxima Interface

To start up **Xmaxima**, you can use the Windows Start Menu route

```
Start, All Programs, Maxima 5.19.0, xmaxima
```

or click on the desktop icon for Xmaxima. You can copy the desktop icon to the Clipboard and paste it into any work folder for quick access.

If you are new to Maxima, go through the quick start tutorial in the bottom window of Xmaxima. You can either click on the expression to have it evaluated, or you can enter the expression, followed by a semi-colon (`;`) in the upper window and press enter to have Maxima carry out the operation. In the top Xmaxima window, note that the two key command **Alt+p** will type in the previous entry, and you can keep entering **Alt+p** until you get the entry you want (to edit or simply rerun).

Once you know what you are doing, you can close the bottom window of Xmaxima by using the Menu bar. On that menu bar, choose **Options, Toggle Browser Visibility**. To use only keyboard commands to close the bottom window, use the succession: **Alt + e**, **RightArrow**, **Enter**.

To quit Xmaxima, choose **File, Exit** on the menu bar (or **Alt+f**, **x**).

The second short introduction can be found on the **Start,All Programs, Maxima 5.19.0, Introduction** link. Written by Cornell University (Dept. of Theoretical and Applied Mechanics) Professor Richard Rand (`http://tam.cornell.edu/`), this is an excellent short introduction to many basic features of Maxima. The advice in Rand's introduction to exit Maxima by typing `quit();` is relevant if you are using the "command line maxima" version, aka "maxima console".

One important thing to remember when using Xmaxima is to never press **Enter** (to execute code) with a space between the semicolon (or the dollar sign) and the postion of the cursor. At least on the author's Windows XP machine, Xmaxima will "hang" and refuse to issue the next input prompt, and you will have to click on **File, Restart** on the Xmaxima menu bar to start a new session. This type of error can creep in easily if you are copying code you have previously saved to a text file, and have an extra space in the file. If you then select, copy, and paste that code fragment into Xmaxima, with the space at the end intact, you should carefully backspace to either the semicolon or the dollar sign before pressing **Enter**. The safest path is to make sure your original text selection for copy does not include a space beyond the dollar sign.

The Maxima Help Manual

The most important continuous source of information about Maxima syntax and reserved words is the Maxima Manual, which you should leave open in a separate window. To open a separate Maxima Manual window from inside the XMaxima interface, click on the XMaxima menu item: **Help, Maxima Manual** (you can use the shortcut **Alt+h** to open the Help menu).

Move around this reference manual via either **Contents** or **Index**. For example, left-click **Index** and start typing `integrate`. By the time you have typed in `inte`, you are close to the `integrate` entry, and you can either continue to type the whole word, or use the down arrow key to select that entry. Then press the **Enter** key. On the right side will be the Maxima Manual entry for `integrate`.

To scroll the Maxima Manual page, double-click on the page, and then use the **PageDown** and **PageUp** keys and the **UpArrow** and **DownArrow** keys. To return to the index entry panel, left click that panel, and type `diff`, which will take you to the section of the Maxima Manual which explains how to evaluate a derivative.

The Xmaxima Manual

If you look at the Xmaxima manual via **Help, Xmaxima Manual (Web Browser)** , your default browser will come up with a somewhat out of date manual with the sections: 1. Command-line options, 2. Xmaxima Window, 3. Entering commands, 4. Session control, 5. Openmath plots, 6. The browser, 7. Getting Help, and Concept Index.

The first section "1. Command-line options" is not relevant for Windows XP Xmaxima.

Xmaxima Font Choices

In Sec 2, Xmaxima Window, you will find the statement:

> You can also choose different types and sizes for the fonts, in the section 'Preferences' of the Options menu; those settings will be saved for future sessions.

The defaults are Times New Roman with size adjustment 3 for proportional fonts and Courier New with size adjustment 2 for fixed fonts. Using the menu with **Options, Preferences**, you can click on the typeface buttons to select a different font type, and can click on the size number button to select another font size. You then should click the **Apply and Quit** button.

Entering Your Expression

In Xmaxima, every input prompt, like **`(%i1)`**, is waiting for an input which conforms to Maxima's syntax expectations. There is no such thing as a "text cell", although you can include text comments anywhere as long as they are delimited by the standard comment delimiters **`/*`** and **`*/`**, which only need to occur at the very beginning and end of the comment, even if the comment extends over many lines.

Here is the beginning of Sec.3, Entering Commands, from the Xmaxima manual. We have replaced some irrelevant or obsolete material with updated instructions.

> Most commonly, you will enter Maxima commands in the last input line that appears on the text Window. That text will be rendered in weak green. If you press **Enter**, without having written a command-termination character (either **`;`** or **`$`**) at the end, the text will remain green and you can continue to write a multi-line command. When you type a command-end character and press the **Enter** key, the text will become light blue and a response from Maxima should appear in light black. You can also use the **UpArrow** or **DownArrow** keys to move to a new line without sending the input for Maxima evaluation yet. If you want to **clear** part of the current input from the beginning to some point, position your cursor at that point (even if the region thereby selected spans several lines) and then use **Edit, Clear input** or the two-key command **Ctrl+u**.
>
> If you move the cursor over the **`(%i1)`** input label, or any other label or output text (in black), you will not be able to type any text there; that feature will prevent you from trying to enter a command in the wrong place, by mistake. If you really want to insert some additional text to modify Maxima's output, and which will not be interpreted by Maxima, you can do that using cut and paste (we will cover that later).
>
> You can also write a new input command for Maxima on top of a previous input line (in blue), for instance, if you do not want to write down again everything but just want to make a slight change. Once you press the **Enter** key, the text you modified will appear at the last input line, as if you had written it down there; the input line you modified will continue the same in Xmaxima's and Maxima's memory, in spite of having changed in the screen.

For example, suppose you entered **a: 45;** in input line **(%i1)**, and something else in **(%i2)**.

```
(%i1) a:45;
(%o1)                              45
(%i2) b:30;
(%o2)                              30
```

You then move up over the **(%i1) a: 45;** and change the **5** to **8**. You can then press **End** to get the cursor at the end of the command, and then press **Enter** to submit the new (edited) command. At that point the screen looks like:

```
(%i1) a:48;
(%o1)                              45
(%i2) b:30;
(%o2)                              30
(%i3) a:48;
(%o3)                              48
```

But if you now enter **(%i1);** as input **(%i4)** and press **Enter**, the output **(%o4)** will be **a: 45**. The screen will now look like:

```
(%i1) a:48;
(%o1)                              45
(%i2) b:30;
(%o2)                              30
(%i3) a:48;
(%o3)                              48
(%i4) (%i1);
(%o4)                            a : 45
```

Maxima knows the current binding of both **(%i1)** (which is the output **(%o4)**) and **a**

```
(%i5) a;
(%o5)                              48
```

If you navigate through the input lines history (see next section), you will also see that the first input keeps its original value.

Speeding up Your Work with XMaxima

When you want to edit your previous command, use **Alt+p** to enter the previous input (or use enough repetitions of **Alt+p** to retrieve the command you want to start with). If the code extends over several screen lines, and/or your editing will include deleting lines, etc., delete the command-completion character at the end (**;** or **$**) first, and then edit, and then restore the command completion character to run the edited code.

The use of the keyboard keys **Home**, **End**, **PageUp**, **PageDown**, **Ctrl+Home**, and **Ctrl+End** (as well as **UpArrow** and **DownArrow** greatly speeds up working with Xmaxima

For example to rerun as is or to copy a command which is located up near the top of your current Xmaxima session, first use **Home** to put the cursor at the **beginning** of the current line, then **PageUp** or **Ctrl+Home** to get up to the top region fast.

If you simply want to rerun that command as is, use **End** to get the cursor at the end of the entry (past the command-completion character), and simply press **Enter** to retry that command. The command will be evaluated with the state of information Maxima has after the last input run. That entry will be automatically entered into your session (with new output) at the bottom of your session screen. You can get back to the bottom using the **Ctrl+End** two-key command.

Alternatively, if you don't want the retry the exact same command, but something similar, then select the part of the code you want to use as a starting point for editing and press **Ctrl+c** to copy your selection to the Window's Clipboard. To select, you can either drag over the code with the mouse while you hold down the left mouse button, or else hold down the **Shift** key with your left hand and and combine with the **End**, the **LeftArrow**, and the **DownArrow** keys to help you select a region to copy.

Once you have selected and copied, press **Ctrl+End** to have the cursor move to the bottom of your workspace where XMaxima is waiting for your next input and press **Ctrl+v** to paste your selection. If the selection extends over multiple lines, use the down cursor key to find the end of the selection. If your selection included the command-completion character, remove it (backspace over that final symbol) before starting to edit your copied selection.

You are then in the driver's seat and can move around the code and make any changes without danger of Xmaxima preemptively sending your work to the Maxima engine until you finally have completed your editing, and move to the very end and provide the proper ending (either `;` or `$`) and then press **Enter** to evaluate the entry.

Using the Input Lines History

If your cursor is positioned next to the active input prompt at the bottom of the screen (ie., where **Ctrl+End** places the cursor), you can use the key combinations **Alt+p** and **Alt+n** to recover the previous or next command that you entered. For example, if you press **Alt+n**, you will enter the first input **`(%i1)`**, and if you continue to press **Alt+n**, you will get in sucession **`(%i2)`**, **`(%i3)`**,...

Alternatively, if the active input prompt is **`(%i10)`** and you press **Alt+p** repeatedly, you will get inputs **`(%i9)`**, **`(%i8)`**, **`(%i7)`**, ...

Searching for a String in Your Previous Inputs

Those same two-key combinations can also be used to search for a previous input lines containing a particular string. Suppose you have one or more previous lines that included (among other functions) **sin(something)**. At the last input prompt, type **sin** and then use either of the two-key commands **Alt+p** or **Alt+n** repeatedly until the particular input line containing the instance of **sin** you are looking for appears. You can then either immediately rerun that input line (press **End** to get the cursor at the end of the input and then press **Enter**) or you can edit the input line using **RightArrow** and **LeftArrow**, **Home**, and **End** to move around, and finally complete your editing and press **Enter**. In summary, you first write down the string to search, and then **Alt+p**, to search backwards, or **Alt+n** to search forward. Pressing those key combinations repeatedly will allow you to cycle through all the lines that contain the string. If you want to try a different string in the middle of the search, you can delete the current input, type the new string, and start the search again.

Cutting and Pasting

You can cut or copy a piece of text that you select, from anywhere on the text window (ie., the main top window of Xmaxima); not only from the input lines but also from the output text in black. To select the text, you can drag the cursor with the mouse while you keep its left button depressed, or you can hold the **Shift** key with one finger, while you move the cursor with the mouse or with the arrow keys.

Once you have selected the text, you can either cut it, with **Edit, cut** or the shortcut **Ctrl+x**, **or** copy it to an internal buffer using **Edit, copy** or **Ctrl+c**. The text that has been cut or copied most recently can be pasted anywhere, even in the output fields, using **Edit, paste** or **Ctrl+v**.

There is a command similar to 'cut', called 'kill', accessed via either **Edit, kill** or **Ctrl+k**, with two major differences: it only works in input fields (blue or green) and instead of cutting a text that was selected, it will cut all the text from the cursor until the **end** of the input line.

The command **Edit, Clear input** or **Ctrl+u** is similar to **Edit, kill**, but it will only work on the last input line (ie the current input line) and will clear all from the beginning of that input line to the cursor position.

To paste the **last** text that you have cut with either 'kill' or 'clear input', you should use the 'yank' command **Edit, yank** or **Ctrl+y**. If you use the Clear Input command, **Ctrl+u**, you can immediately restore the line with the yank command **Ctrl+y** in a sort of "UnDo".

Other Keyboard Shortcuts

Other useful key combinations are:
Ctrl+f, the same as **RightArrow**,
Ctrl+b, the same as **LeftArrow**,
Ctrl+p, the same as **UpArrow**,
Ctrl+n, the same as **DownArrow**,
Either **Ctrl+a** or **Home** moves to the left end of the current line (to the left of `(%xn)`),
Either **Ctrl+e** or **End** moves to the right end of the current line,
Ctrl+Home moves to the first character at the top of the text window,
Ctrl+End, moves to the last character at the bottom of the text window.

Save Xmaxima Session Record as a Text File

The menu command **Edit, Save Console to File** will bring up a dialog which allows you to select the folder and file name with the default extension **.out**, and Xmaxima will save the current session screen, as it appears to you, to a text file with that name. This can be a convenient way to keep a record of your work, particularly if you use the day's date as part of the name. You can then open that text file with any text editor, such as **Notepad2**, and edit it as usual, and you can also change the name of the file. This session record is not in the form of inputs which you could use immediately with Xmaxima, although you could copy and paste command inputs one at a time into a later Xmaxima session.

Save All Xmaxima Inputs as Lisp Code for Later Use in Maxima

The menu command **File, Save Expressions to File** will open a dialog which will save every input as Lisp code with a file name like **sat.bin** or **sat.sav**. Although you can read such a file with a normal text editor, its main use to to rerun all the inputs in a later session by using **load("sat.bin")**, for example. All the variable assignments and function definitions will be absorbed by Maxima, but you don't see any special output on the screen. Instead of that menu route, you could just type the input `save("C:/work2/sat1.bin",all);` to create the Lisp code record of session inputs.

Save All Xmaxima Inputs in a Batch File Using stringout

If you type the input `stringout("c:/work2/sat1.mac",input);`, the inputs will be saved in the form of Maxima syntax which can later be batched into a later Maxima session. In a later session, you type the input `batch("c:/work2/sat1.mac");` and on the screen will appear each input line as well as what each output is in detail. (Or you could use the menu route **File, Batch File**, which will open a dialog which lets you select the file).

Quiet Batch Input

If you use the menu route **File, Batch File Silently**, you will not see the record of inputs and outputs in detail, but all the bindings and definitions will be absorbed by Maxima. There will be no return value and no indication which file has been loaded, so you might prefer typing `load("c:/work2/sat1.mac");`, or `batchload("c:/work2/sat1.mac");` just to have that record in your work.

1.5 Creating and Using a Startup File: maxima-init.mac

You can create a startup file which will be read by Maxima at the start (or restart) of a new Maxima session. If you either have not already created your startup file or have not interactively changed the binding of **maxima_userdir**, you can find where Maxima expects to find a startup file as follows. We purposely start Xmaxima from a link to **..bin/xmaxima.exe** on the desktop, so Xmaxima has no clue where our work folder is.

```
(%i1) maxima_userdir;
(%o1)           C:/Documents and Settings/Edwin Woollett/maxima
```

Before we show the advantages of using your startup file, let's show what you need to do to load one of your ***.mac** files in your work folder into Maxima without that startup file helping out. In my work folder **c:/work2** is a file **qfft.mac**, (available with Ch. 11 material on the author's web page) and here is an effort to load the file in Xmaxima.

```
(%i2) load(qfft);
Could not find `qfft' using paths in file_search_maxima,file_search_lisp.
 -- an error.  To debug this try debugmode(true);
(%i3) load("qfft.mac");
Could not find `qfft.mac' using paths in file_search_maxima,file_search_lisp.
 -- an error.  To debug this try debugmode(true);
(%i4) load("c:/work2/qfft.mac");
 type qfft_syntax(); to see qfft and qift syntax
(%o4)                        c:/work2/qfft.mac
```

On the other hand, if we have a link to **..bin/xmaxima.exe** sitting in our work folder **c:/work2** and we start Xmaxima using that folder link we get

```
(%i2) load(qfft);
Could not find `qfft' using paths in file_search_maxima,file_search_lisp.
 -- an error.  To debug this try debugmode(true);
(%i3) load("qfft.mac");
 type qfft_syntax(); to see qfft and qift syntax
(%o3)                             qfft.mac
```

which shows one of the virtues of starting up Xmaxima using a link in your work folder.

Now that you know where Maxima is going to look for your startup file, make sure such a folder exists and create a file named **maxima-init.mac** in that folder. You can have Maxima display a simple message when it reads your startup file as an added check on what exactly is going on. For example, you could have the line **disp ("Hi, Cindy")$**.

After saving the current version of that file in the correct folder, that message should appear after the Maxima version and credits message.

Below is the author's startup file, in which he has chosen to tell Maxima to look in the **c:/work2** folder for `*.mac` or `*.mc` files, as well as the usual search paths. The chapter 1 utility file **mbe1util.mac** is loaded into the session.

```
/* this is c:\Documents and Settings\Edwin Woollett\maxima\maxima-init.mac  */
/* last edit: 7-28-09  */
maxima_userdir: "c:/work2" $
maxima_tempdir : "c:/work2"$
file_search_maxima : append(["c:/work2/###.{mac,mc}"],file_search_maxima )$
file_search_lisp : append(["c:/work2/###.lisp"],file_search_lisp )$
load(mbe1util)$
print("   mbe1util.mac functions ", functions)$
disp("Maxima is the Future!")$
```

Naturally, you need to replace `c:/work2` by the path to your own work folder.

With this startup file in place, here is the opening screen of Maxima, using the link to Xmaxima in my work folder to start the session, and setting input **(%i1)** to be a request for the binding of **maxima_userdir**:

```
Maxima 5.19.0 http://maxima.sourceforge.net
Using Lisp GNU Common Lisp (GCL) GCL 2.6.8 (aka GCL)
Distributed under the GNU Public License. See the file COPYING.
Dedicated to the memory of William Schelter.
The function bug_report() provides bug reporting information.
   mbe1util.mac functions  [qplot(exprlist, prange, [hvrange]), rtt(e),
ts(e, v), to_atan(e, y, x), to_atan2(e, y, x), totan(e, v), totan2(e, v),
mstate(), mattrib(), mclean(), fll(x) ]
                             Maxima is the Future!

(%i1) maxima_userdir;
(%o1)                              c:/work2
```

Now it is easy to load in the package **qfft.mac**, and see the large increase in the number of user defined functons.

```
(%i2) load(qfft)$
(%i3) functions;
(%o3) [qplot(exprlist, prange, [hvrange]), rtt(e), ts(e, v), to_atan(e, y, x),
to_atan2(e, y, x), totan(e, v), totan2(e, v), mstate(), mattrib(), mclean(),
fll(x), nyquist(ns, fs), sample(expr, var, ns, dvar), vf(flist, dvar),
current_small(), setsmall(val), _chop%(ex), fchop(expr), fchop1(expr, small),
_fabs%(e), kg(glist), spectrum1(glist, nlw, ymax),
spectrum(glist, nlw, ymax, [klim]), spectrum_eps(glist, nlw, ymax, fname,
[klim])]
```

We can use the utility function **mclean()** to remove those **qfft** functions.

```
(%i4) mclean();
----- clean start

(%o0)
(%i1) functions;
(%o1) [qplot(exprlist, prange, [hvrange]), rtt(e), ts(e, v), to_atan(e, y, x),
to_atan2(e, y, x), totan(e, v), totan2(e, v), mstate(), mattrib(), mclean(),
fll(x)]
```

If we had instead used **kill (all)** , the intial set of utilities loaded in from **mbe1util.mac** would have also vanished.

```
(%i2) kill(all);
(%o0)                                done
(%i1) functions;
(%o1)                                 []
```

Of course you could then use a separate **load(mbe1util)** command to get them back.

1.6 Maxima Expressions, Numbers, Operators, Constants and Reserved Words

The basic unit of information in Maxima is the expression. An expression is made up of a combination of operators, numbers, variables, and constants. Variables should have names which are not reserved words (see below), and can represent any type of data structue; there is no requirement to "declare" a variable to be of a certain type.

Numbers in Maxima

Maxima uses

- Integers, such as **123456**,
- Rational numbers, such as $\mathbf{3/2}$, ratios of integers,
- Floats and bigfloats such as **1.234**, **1.234e-6**, and **1.234b5**,
- Complex numbers, such as **4 + 2*%i** and **a + b*%i**. Maxima assumes the symbols **a** and **b** represent real numbers by default.

Operators in Maxima

The table below lists some Maxima operators in order of priority, from lowest to highest.
The input **x^2+3** means $\mathbf{x^2 + 3}$, and not $\mathbf{x^{2+3}}$. The exponentiation has higher precedence than addition. The input **2^3^4** stands for **2^(3^4)**. Parentheses can be used to force order of operations, or simply for clarity.

```
(%i1) x^2+3;
                                    2
(%o1)                              x  + 3
(%i2) 2^3^4;
(%o2)                    2417851639229258349412352
(%i3) 2^(3^4);
(%o3)                    2417851639229258349412352
(%i4) (2^3)^4;
(%o4)                               4096
```

Since the operator $*$ has precedence over $+$, $\mathbf{a + b * c}$ means $\mathbf{a + (b * c)}$, rather than $\mathbf{(a + b) * c}$.

Operator	Description
$+$	addition
$-$	subtraction
$*$	multiplication
$/$	division
$-$	negation
^	exponentiation
.	non-commutative multiplication
^^	non-commutative exponentiation
!	factorial
!!	double factorial

Table 1: Operators in Order of Increasing Precedence

Constants in Maxima

The following table summarizes predefined constants.

Constant	Description
%e	Base of the natural logarithms ($\mathbf{e}$)
%i	The square root of (-1) $(\mathbf{i})$
%pi	The transcendental constant pi ($\boldsymbol{\pi}$)
%phi	The golden mean $(\mathbf{1}+\sqrt{\mathbf{5}})/\mathbf{2}$
%gamma	The Euler-Mascheroni constant
inf	Real positive infinity (∞)
minf	Real negative infinity ($-\infty$)

Table 2: Maxima Predefined Constants

Here are the numerical values to 16 digit precision.

```
(%i5) float( [%e,%pi,%phi,%gamma] );
(%o5) [2.718281828459045, 3.141592653589793, 1.618033988749895,
                                                    0.57721566490153]
```

Reserved Words

There are a number of reserved words which, if used as variable or function names, might be confusing to both the user and Maxima. Their use might cause a possibly cryptic syntax error. Here are some of the "well known" and "less well known but short" reserved words Of course there are many other Maxima function names, global option variables, and

af	else	ic2	plog
and	elseif	if	psi
av	erf	ift	product
args	ev	ilt	put
array	exp	in	rat
at	f90	ind	rem
bc2	fft	inf	rk
carg	fib	inrt	some
cf	fix	integrate	step
cint	for	is	sum
col	from	li	then
cov	gcd	limit	thru
cv	gd	min	und
del	get	next	unless
diag	go	not	vers
diff	hav	op	while
do	ic1	or	zeta

Table 3: Some Simple Reserved Words

system option variables which you might also try to avoid when naming your own variables and function. One quick way to check on "name conflicts" is to keep the html Maxima help manual up in a separate window, and have the Index panel available to type in a name you want to use. The painful way to check on name conflicts is to wait for Maxima to give you a strange error message as to why what you are trying to do won't work. If you get strange results, try changing the names of your variables and or functions.

An important fact is that Maxima is case sensitive, so you can avoid name conflicts by capitalizing the names of your user defined Maxima functions. Your **Solve** will not conflict with Maxima's **solve**. This is a dumb example, but illustrates the principle:

```
(%i6) Solve(x):= x^2;
                                                 2
(%o6)                           Solve(x) := x
(%i7) Solve(3);
(%o7)                                  9
```

Of course, it takes more typing effort to use capitalized function names, which is why they are not popular among power users.

1.7 Input and Output Examples

In the following we are using the **Xmaxima** interface.

As discussed in Sec. 1.3.1, a rational expression is a ratio of polynomials or a sum of such. A special case is a polynomial. A rational expression is a special case of what is called an expression in Maxima.

Here we write a rational expression without binding it to any particular symbol.

```
(%i1) x/(x^3+1);
                                       x
(%o1)                                ------
                                      3
                                     x  + 1
```

The input line is typed in a "one dimensional" version and, if the input completion character is a semi-colon `;` then Xmaxima displays the expression in a text based two-dimensional notation. You can force one-dimensional output if you set **display2d** to **false**:

```
(%i2) display2d:false$
(%i3) %o1;
(%o3) x/(x^3+1)
(%i4) display2d:true$
(%i5) %o1;
                                       x
(%o5)                                ------
                                      3
                                     x  + 1
```

When you want to show a piece of code to the Maxima mailing list, it is recommended that you show the output in the one-dimensional form since otherwise, in the received message, exponents can appear in a shifted position which may be hard to interpret.

The output `(%o6)` is a symbolic expression which can be manipulated as a data structure. For example, you can add it to itself. The Maxima symbol `%` refers to the last output line.

```
(%i6) % + %;
                                      2 x
(%o6)                                ------
                                      3
                                     x  + 1
```

Notice the automatic simplification carried out by default. Maxima could have left the result as a sum of two terms, but instead recognised that the summands were identical and added them together producing a one term result. Maxima will automatically (in default behavior) perform many such simplifications.

Here is an example of automatic simplification of a trig function:

```
(%i7) sin(x - %pi/2);
(%o7)                               - cos(x)
```

1.8 Maxima Power Tools at Work

1.8.1 The Functions apropos and describe

Function apropos

apropos("foo") returns a list of core Maxima names which have **foo** appearing anywhere within them. For example, **apropos ("exp")** returns a list of all the core flags and functions which have **exp** as part of their names, such as **expand**, **exp**, and **ratexpand**. Thus if you can only remember part of the name of something, you can use this command to find the correct complete name. Here is an example:

```
(%i1) apropos ("exp");
(%o1) [askexp, auto_mexpr, besselexpand, beta_expand, cfexpand, comexp,
domxexpt, dotexptsimp, errexp, errexp1, errexp2, errexp3, exp, exp-form,
expand, expandwrt, expandwrt_denom, expandwrt_factored, expandwrt_nonrat,
expansion, expint, expintegral_chi, expintegral_ci, expintegral_e,
expintegral_e1, expintegral_ei, expintegral_hyp, expintegral_li,
expintegral_shi, expintegral_si, expintegral_trig, expintexpand, expintrep,
explicit, explose, expon, exponentialize, expop, expr, exprlist, expt,
exptdispflag, exptisolate, exptsubst, Expt, facexpand, factorial_expand,
gamma_expand, logexpand, macroexpand, macroexpand1, macroexpansion, matrixexp,
poisexpt, psexpand, radexpand, ratexpand, ratsimpexpons, sexplode,
solveexplicit, sumexpand, taylor_logexpand, texput, trigexpand,
trigexpandplus, trigexpandtimes, tr_exponent, tr_warn_fexpr]
```

Function describe

describe(e), describe(e, exact), describe(e, inexact)
describe(e) is equivalent to **describe(e, exact)** and prints to the screen the manual documentation of **e**.

describe(e, inexact) prints to the screen a numbered list of all items documented in the manual which contain "e" as part of their name. If there is more than one list element, Maxima asks the user to select an element or elements to display.

SHORTCUTS:
At the interactive prompt, **? foo** (with a space between **?** and **foo**) and NO ENDING SEMICOLON (just press Enter) is equivalent to either **describe(foo)** or **describe(foo, exact)**, and **?? foo** (with a space between **??** and **foo**) and NO ENDING SEMICOLON (just press Enter) is equivalent to **describe(foo, inexact)**.

In the latter case, the user will be asked to type either a set of space separated numbers to select some of the list elements, such as **2 3 5** followed by Enter, or the word **all** followed by Enter, or the word **none** followed by Enter.

Here is an example of interactive use of the single question mark shortcut.

```
(%i2) ? exp
 -- Function: exp (<x>)
     Represents the exponential function.  Instances of `exp (<x>)' in
     input are simplified to `%e^<x>'; `exp' does not appear in
     simplified expressions.

     `demoivre' if `true' causes `%e^(a + b %i)' to simplify to `%e^(a
     (cos(b) + %i sin(b)))' if `b' is free of `%i'. See `demoivre'.

     `%emode', when `true', causes `%e^(%pi %i x)' to be simplified.
     See `%emode'.

     `%enumer', when `true' causes `%e' to be replaced by 2.718...
     whenever `numer' is `true'. See `%enumer'.
```

```
  There are also some inexact matches for `exp'.
  Try `?? exp' to see them.

(%o2)                                true
```

1.8.2 The Function ev and the Properties evflag and evfun

Function ev

ev is a "jack-of-all-trades swiss army knife" which is frequently useful, occasionally dangerous, and complex to describe.

In brief, the syntax is

```
     ev ( expr, options );
        or more explicitly,
     ev (expr, arg_1, ..., arg_n)
```

with the interactive mode shortcut

```
     expr, options ;
       or again more explicitly
     expr, arg_1, ..., arg_n ;
```

The interactive mode shortcut form cannot be used inside user defined Maxima functions or blocks.

The Manual description of **ev** begins

```
Evaluates the expression expr in the environment specified by the
  arguments arg_1, ..., arg_n. The arguments are switches (Boolean flags),
  assignments, equations, and [Maxima] functions. ev returns the result
  (another expression) of the evaluation.
```

One option has the form `V : e`, or `V = e`, which causes `V` to be bound to the value of `e` during the evaluation of `expr`. If more than one argument to **ev** is of this type, then the binding is done in parallel, as shown in the following example.

```
(%i1) x+y, x  =  a+y;
(%o1)                           2 y + a
(%i2) %, y = 2;
(%o2)                            a + 4
(%i3) x+y, x = a+y, y = 2;
(%o3)                          y + a + 2
(%i4) x+y, [x = a+y, y = 2];
(%o4)                          y + a + 2
```

If `V` is a non-atomic expression, then a substitution rather than a binding is performed.

This example illustrates the subtlety of the way **ev** in designed to work, and shows that some experimentation should be carried out to gain confidence in the result returned by **ev**.

The next example of **ev** shows its use to check the correctness of solutions returned by **solve**.

```
(%i1) eqns : [-2*x -3*y = 3, -3*x +2*y = -4]$
(%i2) solns : solve (eqns);
                                  17      6
(%o2)                      [[y = - --, x = --]]
                                  13      13
(%i3) eqns, solns;
(%o3)                       [3 = 3, - 4 = - 4]
```

Our final example shows the use of **rectform** and **ratsimp** to make explicit the fourth roots of -1 returned by **solve**.

```
(%i1) solve ( a^4 + 1 );
                 1/4                  1/4              1/4                1/4
(%o1) [a = (- 1)    %i, a = - (- 1)   , a = - (- 1)    %i, a = (- 1)   ]
(%i2) % , rectform, ratsimp;
          sqrt(2) %i - sqrt(2)         sqrt(2) %i + sqrt(2)
(%o2) [a = --------------------, a = - --------------------,
                   2                            2
                         sqrt(2) %i - sqrt(2)      sqrt(2) %i + sqrt(2)
                   a = - --------------------, a = --------------------]
                                  2                         2
(%i3) %^4, ratsimp;
                        4          4          4          4
(%o3)                 [a  = - 1, a  = - 1, a  = - 1, a  = - 1]
```

Both **rectform** and **ratsimp** are Maxima functions which have the property **evfun** (see next entry), which means that **`ev(expr, rectform, ratsimp)`** is equivalent to **`ratsimp ( rectform ( ev(expr) ) )`** .

Property evflag

When a symbol **`x`** has the **evflag** property, the expressions **`ev(expr, x)`** and **`expr, x`** (at the interactive prompt) are equivalent to **`ev(expr, x = true)`**. That is, **`x`** is bound to **true** while **`expr`** is evaluated.
The expression **`declare(x, evflag)`** gives the **evflag** property to the variable **`x`**.
The flags which have the **evflag** property by default are the following:

```
algebraic, cauchysum, demoivre, dotscrules, %emode, %enumer, exponentialize,
exptisolate, factorflag, float, halfangles, infeval, isolate_wrt_times,
keepfloat, letrat, listarith, logabs, logarc, logexpand, lognegint, lognumer,
m1pbranch, numer_pbranch, programmode, radexpand, ratalgdenom, ratfac,
ratmx, ratsimpexpons, simp, simpsum, sumexpand, and trigexpand.
```

Even though the Boolean switch **numer** does not have the property **evflag**, it can be used as if it does.

```
(%i4) [exponentialize,float,numer,simp];
(%o4)                    [false, false, false, true]
(%i5) properties(exponentialize);
(%o5) [system value, transfun, transfun, transfun, transfun, transfun,
                                                         transfun, evflag]
(%i6) properties(numer);
(%o6)                  [system value, assign property]
(%i7) properties(float);
(%o7) [system value, transfun, transfun, transfun, transfun, transfun,
                                               evflag, transfun, transfun]
(%i8) properties(simp);
(%o8)                      [system value, evflag]
```

Here are some examples of use.

```
(%i9) [ ev (exp(3/29), numer ), ev (exp(3/29), float) ];
(%o9)                 [1.108988430411017, 1.108988430411017]
(%i10) [ ev (exp (%pi*3/29), numer), ev (exp (%pi*3/29), float) ];
                                          0.10344827586207 %pi
(%o10)              [1.384020049155809, %e                    ]
(%i11) 2*cos(w*t) + 3*sin(w*t), exponentialize, expand;
                       %i t w                       - %i t w
            3 %i %e            %i t w    3 %i %e              - %i t w
(%o11)    - ------------- + %e       + --------------- + %e
                  2                            2
```

(For the use of the “key word” **expand** in this context, see below.)

Property evfun

When a Maxima function **`F`** has the **evfun** property, the expressions **`ev(expr, F)`** and **`expr, F`** (at the interactive prompt) are equivalent to **`F ( ev (expr))`**.
If two or more **evfun** functions **`F`**, **`G`**, etc., are specified, then **ev (expr, F, G)** is equivalent to **G (F (ev(expr)))**.

The command **`declare(F, evfun)`** gives the **evfun** property to the function **`F`**.
The functions which have the **evfun** property by default are the following very useful and single argument funtions: **bfloat, factor, fullratsimp, logcontract**, **polarform, radcan, ratexpand, ratsimp, rectform**, **rootscontract, trigexpand, and trigreduce**.

Note that **rat**, and **trigsimp** do not, by default, have the property **evfun**.

Some of the other "key words" which can be used with **ev** are **expand, nouns, diff, integrate** (even though they are not obviously boolean switches and do not have the property **evflag**).

```
(%i12) [diff,expand,integrate,nouns];
(%o12)                  [diff, expand, integrate, nouns]
(%i13) properties(expand);
(%o13)         [transfun, transfun, transfun, transfun, transfun]
(%i14) (a+b)*(c+d);
(%o14)                           (b + a) (d + c)
(%i15) (a+b)*(c+d),expand;
(%o15)                       b d + a d + b c + a c
(%i16) ev((a+b)*(c+d),expand);
(%o16)                       b d + a d + b c + a c
```

The undocumented property **transfun** apparently has something to do with translation to Lisp.

1.8.3 The List functions and the Function fundef

functions

functions is the name of a list maintained by Maxima and this list is printed to the screen (as would any list) with the names of the current available non-core Maxima functions (either user defined interactively or defined by any packages loaded into the current session). Here is the example we saw previously based on the author's startup file:

```
(%i1) functions;
(%o1) [qplot(exprlist, prange, [hvrange]), rtt(e), ts(e, v), to_atan(e, y, x),
to_atan2(e, y, x), totan(e, v), totan2(e, v), mstate(), mattrib(), mclean(),
fll(x)]
```

fundef

fundef(name) prints out the definition of a non-core Maxima function (if currently known by Maxima). Here is an example related to the previous example which said that Maxima knew about a function called **rtt**.

```
(%i2) fundef(rtt);
(%o2)                rtt(e) := radcan(trigrat(trigsimp(e)))
(%i3) fundef(cos);
cos is not the name of a user function.
 -- an error.  To debug this try debugmode(true);
```

1.8.4 The Function kill and the List values

kill

kill(a,b) will eliminate the objects **a** and **b**. Special cases are **kill(all)** and **kill(allbut(x,y))**. Here is an example which removes our user defined Maxima function **rtt**.

```
(%i1) kill(rtt);
(%o1)                                done
(%i2) functions;
(%o2) [qplot(exprlist, prange, [hvrange]), ts(e, v), to_atan(e, y, x),
to_atan2(e, y, x), totan(e, v), totan2(e, v), mstate(), mattrib(), mclean(),
fll(x)]
```

values

values is another list maintained by Maxima which contains the names of currently assigned scalar values which have been set by the user interactively or by packages which have been loaded. Here is a simple example.

```
(%i3) [ a:2, b:5, e: x^2/3 ];
                                          2
                                         x
(%o3)                            [2, 5, --]
                                         3
(%i4) values;
(%o4)                             [a, b, e]
```

1.8.5 Examples of map, fullmap, apply, grind, and args

Let `f` be an unbound symbol. When `f` is then mapped onto a list or expression, we can see what will happen if we **map** a Maxima function on to the same kind of object.

```
(%i1) map('f, [x, y, z] );
(%o1)                        [f(x), f(y), f(z)]
(%i2) map('f, x + y + z);
(%o2)                        f(z) + f(y) + f(x)
(%i3) map('f, [a*x,b*exp(y),c*log(z)]);
                                     y
(%o3)                  [f(a x), f(b %e ), f(c log(z))]
(%i4) map('f, a*x + b*exp(y) + c*log(z) );
                                          y
(%o4)                  f(c log(z)) + f(b %e ) + f(a x)
```

The use of **map** with the Maxima function **ratsimp** allows a separate simplification to be carried out on each term of the following expression.

```
(%i5) e : x/(x^2+x)+(y^2+y)/y ;
                                 2
                                y  + y     x
(%o5)                           ------ + ------
                                  y       2
                                         x  + x
(%i6) e, ratsimp;
                               (x + 1) y + x + 2
(%o6)                          -----------------
                                     x + 1
(%i7) map('ratsimp,e);
                                       1
(%o7)                            y + ----- + 1
                                     x + 1
```

We create a list of equations from two lists using **map**.

```
(%i8) map( "=", [x,y,z],[a,b,c] );
(%o8)                          [x = a, y = b, z = c]
```

We compare **map** and **fullmap** when appied to an expression.

```
(%i9) expr : 2*%pi + 3*exp(-4);
                                              - 4
(%o9)                         2 %pi + 3 %e
(%i10) map('f, expr);
                                                - 4
(%o10)                      f(2 %pi) + f(3 %e    )
(%i11) fullmap('f, expr);
                                                  f(- 4)
(%o11)                 f(2) f(%pi) + f(3) f(%e)
```

apply

The Maxima function **apply** can be used with a list only (not an expression).

```
(%i12) apply('f, [x,y,z]);
(%o12)                            f(x, y, z)
(%i13) apply("+",[x,y,z]);
(%o13)                             z + y + x
(%i14) dataL : [ [1,2], [2,4] ]$
(%i15) dataM : apply('matrix, dataL );
                                   [ 1  2 ]
(%o15)                              [      ]
                                   [ 2  4 ]
(%i16) grind(%)$
matrix([1,2],[2,4])$
```

The function **grind** allows one dimensional display suitable for copying into another input line. The last example above created a Maxima **matrix** object from a nested list. Maxima has many tools to work with **matrix** objects. To achieve one dimensional display of such **matrix** objects (saving space), you can use `display2d:false`. To go from a Maxima **matrix** object to a nested list, use **args**.

```
(%i17) args(dataM);
(%o17)                          [[1, 2], [2, 4]]
```

1.8.6 Examples of subst, ratsubst, part, and substpart

subst

To replace `x` by `a` in an expression `expr`, you can use either of two forms.

```
   subst ( a, x, expr );
     or
   subst ( x = a, expr );
```

To replace `x` by `a` and also `y` by `b` one can use

```
  subst ( [ x = a, y = b ], expr );
```

`x` and `y` must be atoms (ie., a number, a string, or a symbol) or a complete subexpression of `expr`. When `x` does not have these characteristics, use `ratsubst( a, x, expr )` instead.

Some examples:

```
(%i1) e : f*x^3 + g*cos(x);
                                             3
(%o1)                          g cos(x) + f x
(%i2) subst ( x = a, e );
                                           3
(%o2)                          cos(a) g + a  f
(%i3) e1 : subst ( x = a + b, e );
                                                3
(%o3)                     cos(b + a) g + (b + a)  f
(%i4) e2 : subst ( a + b = y, e1 );
                                             3
(%o4)                          g cos(y) + f y
(%i5) e3 : f*x^3 + g*cos(y);
                                             3
(%o5)                          g cos(y) + f x
(%i6) e4 : subst ( [x = a+b, y = c+d],e3 );
                                                3
(%o6)                     cos(d + c) g + (b + a)  f
(%i7) subst ([a+b = r, c+d = p],e4 );
                                  3
(%o7)                          f r  + g cos(p)
```

ratsubst

```
(%i8) e : a*f(y) + b*g(x);
(%o8)                          a f(y) + b g(x)
(%i9) e1 : ratsubst( cos(y),f(y),e );
(%o9)                         a cos(y) + b g(x)
(%i10) e2 : ratsubst( x^3*sin(x),g(x),e1 );
                                         3
(%o10)                     a cos(y) + b x  sin(x)
```

part and substpart

The Maxima functions **part** and **substpart** are best defined by a simple example.

```
(%i11) e : a*log(f(y))/(b*exp(f(y)));
                                 - f(y)
                            a %e       log(f(y))
(%o11)                       --------------------
                                     b
(%i12) length(e);
(%o12)                                 2
(%i13) [part(e,0),part(e,1),part(e,2)];
                                 - f(y)
(%o13)                    [/, a %e       log(f(y)), b]
(%i14) substpart("+",e,0);
                               - f(y)
(%o14)                     a %e       log(f(y)) + b
(%i15) length(part(e,1));
(%o15)                                 3
(%i16) [part(e,1,0),part(e,1,1),part(e,1,2),part(e,1,3)];
                                   - f(y)
(%o16)                     [*, a, %e      , log(f(y))]
(%i17) length(part(e,1,3));
(%o17)                                 1
(%i18) [part(e,1,3,0),part(e,1,3,1)];
(%o18)                             [log, f(y)]
```

```
(%i19) substpart(sin,e,1,3,0);
                                    - f(y)
                              a %e       sin(f(y))
(%o19)                        --------------------
                                       b
(%i20) length( part(e,1,3,1) );
(%o20)                                  1
(%i21) [part(e,1,3,1,0), part(e,1,3,1,1)];
(%o21)                               [f, y]
(%i22) substpart(x,e,1,3,1,1);
                                              - f(y)
                              a log(f(x)) %e
(%o22)                        --------------------
                                       b
```

By judicious use of **part** and **substpart**, we see that we can modify a given expression in all possible ways.

1.8.7 Examples of coeff, ratcoef, and collectterms

collectterms

Use the syntax

```
    collectterms ( expr, arg1, arg2, ...)
```

This Maxima function is best explained with an example. Consider the expression:

```
(%i1) ex1 :  a1*(b + c/2)^2 + a2*(d + e/3)^3 , expand;
              3          2                             2
          a2 e     a2 d e         2          3    a1 c                2
(%o1)     ----- + ------- + a2 d  e + a2 d  + ----- + a1 b c + a1 b
           27         3                              4
```

How can we return this expanded expression to the original form? We first use **collectterms**, and then **factor** with **map**.

```
(%i2) collectterms(ex1,a1,a2);
                         3      2                            2
                        e     d e      2       3         c             2
(%o2)               a2 (-- + ---- + d  e + d ) + a1 (-- + b c + b )
                        27     3                         4
(%i3) map('factor, %);
                                           3               2
                                 a2 (e + 3 d)     a1 (c + 2 b)
(%o3)                            ------------- + -------------
                                      27                4
```

Maxima's core simplification rules prevent us from getting the $3^3 = 27$ into the numerator of the first term, and also from getting the $2^2 = 4$ into the numerator of the second term, unless we are willing to do a lot of **substpart** piecework (usually not worth the trouble).

coeff

The syntax

```
     coeff ( expr, x, 3 )
```

will return the coefficient of `x^3` in `expr`.

The syntax

```
coeff ( expr, x )
```

will return the coefficient of **x** in **expr**.
x may be an atom or a complete subexpression of **expr**, such as **sin(y)**, **a[j]**, or **(a+b)**. Sometimes it may be necessary to expand or factor **expr** in order to make **x^n** explicit. This preparation is not done automatically by **coeff**.

```
(%i4) coeff(%, a2);
                                           3
                                  (e + 3 d)
(%o4)                             ----------
                                      27
(%i5) coeff(%, e + 3*d, 3);
                                      1
(%o5)                                 --
                                      27
```

ratcoef (also ratcoeff)

The function **ratcoef** (or **ratcoeff**) has the same syntax as **coeff** (except that **n** should not be negative), but expands and rationally simplifies the expression before finding the coefficient, and thus can produce answers different from **coeff**, which is purely syntactic.

```
(%i6) ex2 : (a*x + b)^2;
                                           2
(%o6)                             (a x + b)
(%i7) coeff(ex2,x);
(%o7)                                 0
(%i8) ratcoeff(ex2,x);
(%o8)                               2 a b
(%i9) ratcoef(ex2,x);
(%o9)                               2 a b
(%i10) ratcoeff(ex2, x, 0);
                                      2
(%o10)                                b
(%i11) ratcoef(ex2, x, 0);
                                      2
(%o11)                                b
```

Both **coeff** and **ratcoef** can be used with equations (as well as expressions).

```
(%i12) eqn : (a*sin(x) + b*cos(x))^3 = c*sin(x)*cos(x);
                                         3
(%o12)              (a sin(x) + b cos(x))   = c cos(x) sin(x)
(%i13) ratcoef(eqn,sin(x),0);
                                 3     3
(%o13)                          b   cos  (x) = 0
(%i14) ratcoef(eqn,sin(x));
                                2     2
(%o14)                     3 a b   cos  (x) = c cos(x)
```

1.8.8 Examples of rat, diff, ratdiff, ratexpand, expand, factor, gfactor and partfrac

Maxima provides tools which allow the user to write an expression in multiple forms. Consider the tenth order polynomial in x given by

```
(%i1) e:(x + 3)^10;
                                            10
(%o1)                               (x + 3)
```

We can use **diff (expr, x)** to find the first derivative of **expr**, and **diff (expr, x, 2)** to find the second derivative of **expr**, and so on.

```
(%i2) de1 : diff ( e, x );
                                              9
(%o2)                             10 (x + 3)
```

Because this expression is a polynomial in **x**, we can also use **ratdiff (expr, x)**:

```
(%i3) de1r : ratdiff ( e, x );
          9        8          7          6            5            4           3
(%o3) 10 x  + 270 x  + 3240 x  + 22680 x  + 102060 x  + 306180 x  + 612360 x
                                                                   2
                                                 + 787320 x  + 590490 x + 196830
(%i4) factor ( de1r );
                                              9
(%o4)                             10 (x + 3)
```

We show three tools for expansion of this polynomial.

```
(%i5) rat(e);
          10       9        8          7          6          5            4
(%o5)/R/ x   + 30 x  + 405 x  + 3240 x  + 17010 x  + 61236 x  + 153090 x
                                                   3           2
                                        + 262440 x  + 295245 x  + 196830 x + 59049
(%i6) ratexpand (e);
       10       9        8          7          6          5            4
(%o6) x   + 30 x  + 405 x  + 3240 x  + 17010 x  + 61236 x  + 153090 x
                                                   3           2
                                        + 262440 x  + 295245 x  + 196830 x + 59049
(%i7) expand (e);
       10       9        8          7          6          5            4
(%o7) x   + 30 x  + 405 x  + 3240 x  + 17010 x  + 61236 x  + 153090 x
                                                   3           2
                                        + 262440 x  + 295245 x  + 196830 x + 59049
(%i8) factor (%);
                                            10
(%o8)                               (x + 3)
```

If you want the lowest powers displayed first, you have to change the setting of **powerdisp** to **true**.

```
(%i9) powerdisp;
(%o9)                                 false
(%i10) powerdisp : true$
(%i11) %o7;
                                  2           3           4          5
(%o11) 59049 + 196830 x + 295245 x  + 262440 x  + 153090 x  + 61236 x
                                                6          7        8       9    10
                                       + 17010 x  + 3240 x  + 405 x  + 30 x  + x
(%i12) powerdisp : false$
```

Next we consider a sum of rational expressions:

```
(%i13) expr: (x - 1)/(x + 1)^2 + 1/(x - 1);
                                x - 1       1
(%o13)                          -------- + -----
                                      2   x - 1
                               (x + 1)
(%i14) expand ( expr );
                           x              1           1
(%o14)                 ------------ - ------------ + -----
                       2              2             x - 1
                      x  + 2 x + 1   x  + 2 x + 1
(%i15) ratexpand ( expr );
                               2
                            2 x                 2
(%o15)                  --------------- + ---------------
                        3    2            3    2
                       x  + x  - x - 1   x  + x  - x - 1
```

We see that **ratexpand** wrote all fractions with a common denominator, whereas **expand** did not. In general, **rat** and **ratexpand** is more efficient at expanding rational expressions. Here is a very large expression with many **%pi**'s which can be simplified easily by the "rat tribe", but **expand** takes a very long time and returns a mess.

```
(%i16) e :
   ((2/%pi-1)*(((%pi/2-1)/(%pi-1)-1)*((6*(2/%pi-2*(4-%pi)/%pi)/%pi-(6*(2*(4-%pi)
     /%pi-(%pi-2)/(%pi/2-1))/(%pi-1)-6*((%pi-2)/(%pi/2-1)-2)*(%pi/2-1)
     /(%pi*(%pi-1)))/(2*((2/%pi-1)*(%pi/2-1)/(2*(%pi-1)) +2)))
      /(((%pi/2-1)/(%pi-1)-1)/(2*((2/%pi-1)*(%pi/2-1)/(2*(%pi-1))+2))+2)
     -(6*(2*(2*%pi-5)/%pi-2/%pi)/%pi-(6*(2/%pi-2*(4-%pi)/%pi)/%pi
     -(6*(2*(4-%pi)/%pi-(%pi-2)/(%pi/2-1))/(%pi-1)-6*((%pi-2)/(%pi/2-1)-2)
     *(%pi/2-1)/(%pi*(%pi-1)))/(2*((2/%pi-1)*(%pi/2-1)/(2*(%pi-1))+2)))
     /(2*(((%pi/2-1)/(%pi-1)-1)/(2*((2/%pi-1)*(%pi/2-1)/(2*(%pi-1))+2))+2)))
     /(2*(2-1/(4*(((%pi/2-1)/(%pi-1)-1)/(2*((2/%pi-1)*(%pi/2-1)/(2*(%pi-1))+2))
     +2)))*(((%pi/2-1)/(%pi-1)-1)/(2*((2/%pi-1)*(%pi/2-1)/(2*(%pi-1))
     +2))+2)))/((2/%pi-1)*(%pi/2-1)/(2*(%pi-1))+2)+(6*(2*(4-%pi)/%pi-(%pi-2)
     /(%pi/2-1))/(%pi-1)-6*((%pi-2)/(%pi/2-1)-2)*(%pi/2-1)/(%pi*(%pi-1)))
     /((2/%pi-1)*(%pi/2-1)/(2*(%pi-1))+2))/2+6*((%pi-2)/(%pi/2-1)-2)/%pi)
     *(x^3-x)/6  + 2*x$
(%i17) ratsimp (e);
              2                   3           3          2
      (128 %pi  - 520 %pi + 528) x  + (194 %pi  - 248 %pi  + 400 %pi - 528) x
(%o17) -----------------------------------------------------------------------
                                  3         2
                            97 %pi  - 60 %pi  - 60 %pi
(%i18) rat (e);
(%o18)/R/
                2                   3           3          2
        (128 %pi  - 520 %pi + 528) x  + (194 %pi  - 248 %pi  + 400 %pi - 528) x
        -----------------------------------------------------------------------
                                    3         2
                              97 %pi  - 60 %pi  - 60 %pi
```

```
(%i19) ratexpand (e);
                        3                         3
                 528 x                  128 %pi x
(%o19) -------------------------- + ---------------------
             3         2                   2
      97 %pi  - 60 %pi  - 60 %pi   97 %pi  - 60 %pi - 60
                3                                                2
          520 x                      528 x                  194 %pi  x
 - --------------------- - -------------------------- + ---------------------
          2                    3         2                    2
   97 %pi  - 60 %pi - 60   97 %pi  - 60 %pi  - 60 %pi   97 %pi  - 60 %pi - 60
         248 %pi x                  400 x
 - --------------------- + ---------------------
          2                    2
   97 %pi  - 60 %pi - 60   97 %pi  - 60 %pi - 60
```

There is a more general form **expand (expr, p, n)** available to control which parts of fractions are expanded and how (see the Manual index entry for **expand**). There is also a more general form for `rat (expr, x_1, ..., x_n)`.

gfactor

The "g" in **gfactor** comes from factorization over the Gaussian integers.

```
(%i20) gfactor ( x^2 + 1 );
(%o20)                         (x - %i) (x + %i)
```

partfrac

We have earlier emphasized the usefulness of **ratsimp**. A Maxima function which in some cases can be considered the "opposite" of **ratsimp** in its results is **partfrac**, which has the syntax `partfrac ( expr, var )`, and which expands `expr` in partial fractions with respect to `var`. Here is the example presented in the Manual.

```
(%i21) e : 1/(1+x)^2 - 2/(1+x) + 2/(2+x);
                                2       2        1
(%o21)                        ----- - ----- + --------
                              x + 2   x + 1          2
                                              (x + 1)
(%i22) e, ratsimp;
                                          x
(%o22)                         - --------------------
                                    3      2
                                   x  + 4 x  + 5 x + 2
(%i23) partfrac (%, x);
                                2       2        1
(%o23)                        ----- - ----- + --------
                              x + 2   x + 1          2
                                              (x + 1)
```

1.8.9 Examples of integrate, assume, facts, and forget

An important symbolic tool is indefinite integration, much of which is performed algorithmically. The original integration package, written by Joel Moses, incorporates the non-algebraic case of the Risch integration algorithm, and the package is called **sin.lisp** (“sin” is a mnemonic from “Symbolic INtegrator”).

Consider the indefinite integral of a rational expression:

```
(%i1) e : x/(x^3 + 1);
                                      x
(%o1)                               ------
                                     3
                                    x  + 1
(%i2) ie : integrate (e, x);
                                        2 x - 1
                     2             atan(-------)
                log(x  - x + 1)         sqrt(3)    log(x + 1)
(%o2)           --------------- + ------------- - ----------
                       6             sqrt(3)          3
```

Indefinite integration can usually be checked by differentiating the result:

```
(%i3) diff (ie, x);
                    2                2 x - 1           1
(%o3)        ------------------ + -------------- - ---------
                        2               2          3 (x + 1)
                (2 x - 1)         6 (x  - x + 1)
             3 (---------- + 1)
                    3
```

This answer, as it stands, is not identical to the starting integrand. The reason is that Maxima merely differentiates term by term. Maxima automatically applies only those simplification rules which are considered “obvious” and “always desirable”. A particular simplification rule might make one expression smaller but might make another expression larger. In such cases, Maxima leaves the decision to the user.

In this example, the starting integrand can be recovered by ”rational simplification” using the **ratsimp** function.

```
(%i4) %, ratsimp;
                                      x
(%o4)                               ------
                                     3
                                    x  + 1
```

We met **ratsimp** and **fullratsimp** (Sec. 1.3.1) in our brief discussion of the buttons and menu system of **wxMaxima**, where the **Simplify** button made use of **ratsimp**, a reflection of the frequent use you will make of this function. You can think of the word **ratsimp** being a mnemonic using the capitalised letters in RATional SIMPlification.

If you try to use **ratdiff** on the above indefinite integration result, you get an obscure error message:

```
(%i5) ratdiff (ie, x);
`ratdiff' variable is embedded in kernel
 -- an error.  To debug this try debugmode(true);
```

The error message is returned because **ratdiff (expr, x)** can only be used for expressions which are either a polynomial in **x** or a ratio of polynomials in **x**.

integrate consults the **assume** database when making algebraic decisions. The content of that database is returned by **facts();**. You can sometimes avoid questions from **integrate** if you first provide some assumptions with **assume**. Here is an example of a definite integral in which we start with no assumptions, and need to answer a question from **integrate**.

```
(%i1) facts();
(%o1)                               []
(%i2) integrate(x*exp(-a*x)*cos(w*x),x,0,inf);
Is  a  positive, negative, or zero?

p;
                                        2    2
                                       w  - a
(%o2)                           - -----------------
                                    4      2  2    4
                                   w  + 2 a  w  + a
```

We can now tell Maxima that the parameter **a** should be treated as a positive number. We thereby avoid the question.

```
(%i3) ( assume(a > 0), facts() );
(%o3)                             [a > 0]
(%i4) integrate(x*exp(-a*x)*cos(w*x),x,0,inf);
                                        2    2
                                       w  - a
(%o4)                           - -----------------
                                    4      2  2    4
                                   w  + 2 a  w  + a
```

You can now tell Maxima to **forget** that assumption.

```
(%i5) (forget(a > 0), facts() );
(%o5)                               []
```

1.8.10 Numerical Integration and Evaluation: float, bfloat, and quad_qags

Let's try a harder integral, this time a definite integral which is expressed by Maxima in terms of a "special function".

```
(%i1) is : integrate( exp(x^3),x,1,2 );
                                    1                           1
      (sqrt(3) %i - 1) (gamma_incomplete(-, - 8) - gamma_incomplete(-, - 1))
                                    3                           3
(%o1) ----------------------------------------------------------------------
                                   6
(%i2) float(is);
(%o2) 0.16666666666667 (- 719.2849028287006 %i
 - 1.0 (0.66608190774162 - 3.486369946257051 %i) - 412.6003937374765)
 (1.732050807568877 %i - 1.0)
(%i3) expand(%);
(%o3)                      275.5109837634787
(%i4) ival:%;
(%o4)                      275.5109837634787
```

You can look up the `gamma_incomplete(a,z)` definition in the Maxima manual, where you will find a reference to the incomplete upper gamma function **A&S 6.5.2**, which refers to Eq. 6.5.2 in **Handbook of Mathematical Functions**, Edited by Milton Abramowitz and Irene A. Stegun, Dover Publications, N.Y., 1965.

We know that since we are integrating along the real x axis from 1 to 2, our integrand in **(%i1)** is real, so that the numerical value of this integral must be a real number. In using **float**, we see that the real answer will be obtained by cancellation of imaginary pieces, so we may have some roundoff errors.

Let's check the accuracy of using **float** here by comparing that answer to the answer returned by the use of **bfloat** together with **fpprec** set to 20.

```
(%i5) bfloat(is),fpprec:20;
(%o5) 1.6666666666666666667b-1 (1.7320508075688772935b0 %i - 1.0b0)
 (- 1.0b0 (6.6608190774161802181b-1 - 3.4863699462570511861b0 %i)
 - 7.1928490282826659929b2 %i - 4.1260039373722578387b2)
(%i6) expand(%),fpprec:20;
(%o6)       5.7824115865893569814b-19 %i + 2.7551098376331160126b2
(%i7) tval20: realpart(%);
(%o7)                    2.7551098376331160126b2
(%i8) abs(ival - tval20);
(%o8)                     1.670912297413452b-10
```

We see that the numerical answer found using **float** only has twelve digit accuracy. Let's look at the numerical values of the `gamma_incomplete` functions involved here as returned by **float**:

```
(%i9) float(gamma_incomplete(1/3,-8));
(%o9)            - 719.2849028287006 %i - 412.6003937374765
(%i10) float(gamma_incomplete(1/3,-1));
(%o10)              0.66608190774162 - 3.486369946257051 %i
```

Finally, let's use a pure numerical integration routine , `quad_qags`, which returns a list consisting of `[answer, est-error, num-integrand-eval, error-code]`.

```
(%i11) quad_qags(exp(x^3),x,1,2);
(%o11)         [275.5109837633116, 3.2305615936931465E-7, 21, 0]
(%i12) abs(first(%) - tval20);
(%o12)                      2.842170943040401b-14
```

and we see that the use of `quad_qags` for a numerical value was more accurate than the comination of **integrate** and **float**.

1.8.11 Taylor and Laurent Series Expansions with taylor

When computing exact symbolic answers is intractable, one can often resort to series approximations to get approximate symbolic results. Maxima has an excellent Taylor series program. As an example, we can get truncated series representations for **sin(x)** and **cos(x)** as follows. According to the Maxima manual

> **taylor (expr, x, a, n)** expands the expression **expr** in a truncated Taylor or Laurent series in the variable **x** around the point $\mathbf{x} = \mathbf{a}$, containing terms through $(\mathbf{x} - \mathbf{a})^{\mathbf{n}}$.

Here we expand around the point $\mathbf{x} = \mathbf{0}$.

```
(%i1) taylor( sin(x),x,0,5 );
                                   3    5
                                  x    x
(%o1)/T/                     x - -- + --- + . . .
                                  6    120
(%i2) taylor( cos(x),x,0,5);
                                   2    4
                                  x    x
(%o2)/T/                      1 - -- + -- + . . .
                                  2    24
```

A truncated series in denoted by the "**/T/**" symbol next to the line label and also by the trailing dots. Maxima retains certain information about such expansions, such as the "quality" of the approximation, so when several series expansions are combined, no terms are computed beyond the degree of the approximation. In our example, our starting expansions are only good through terms of order $\mathbf{x}^{\mathbf{5}}$, so if we multiply the expansions, any terms smaller (in order of magnitude) than

x^5 are neglected.

Here we use **%th(n)**, which refers to the n'th to last line.

```
(%i3) % * %th(2);
                                     3      5
                                  2 x    2 x
(%o3)/T/                      x - ---- + ---- + . . .
                                   3      15
```

The name "taylor" is only an artifact of history, since the Maxima function **taylor** can handle expansions of functions with poles and branch points and automatically generates Laurent series when appropriate as shown here:

```
(%i4) taylor( 1/ (cos(x) - sec(x))^3, x,0,5 );
                                                  2          4
             1     1       11      347    6767 x     15377 x
(%o4)/T/   - -- + ---- + ------ - ----- - ------- - -------- + . . .
              6      4        2   15120   604800    7983360
             x    2 x    120 x
```

1.8.12 Solving Equations: solve, allroots, realroots, and find_root

The **solve** function is described in the Maxima manual

> Function: **solve (expr, x)**
> Function: **solve (expr)**
> Function: **solve ([eqn_1, ..., eqn_n], [x_1, ..., x_n])**
> Solves the algebraic equation **expr** for the variable **x** and returns a list of solution equations for **x**. If **expr** is not an equation, the equation $\mathbf{expr} = \mathbf{0}$ is assumed in its place. **x** may be a function (e.g. f(x)), or other non-atomic expression except a sum or product. **x** may be omitted if **expr** contains only one variable. **expr** may be a rational expression, and may contain trigonometric functions, exponentials, etc.

Example 1

Here we look for solutions of the equation $\mathbf{x^6 = 1}$ or $\mathbf{x^6 - 1 = 0}$. Once we have found candidate solutions, we check them one at a time.

```
(%i1) eqn : x^6 - 1 = 0$
(%i2) solns : solve(eqn);
         sqrt(3) %i + 1      sqrt(3) %i - 1                  sqrt(3) %i + 1
(%o2) [x = --------------, x = --------------, x = - 1, x = - --------------,
               2                   2                               2
                                                        sqrt(3) %i - 1
                                                  x = - --------------, x = 1]
                                                              2
(%i3) for i thru length(solns) do
        disp ( ev( eqn, solns[i], ratsimp ) )$
                                     0 = 0
                                     0 = 0
                                     0 = 0
                                     0 = 0
                                     0 = 0
                                     0 = 0
```

One often uses **solve** in the hope of finding useful symbolic expressions for roots of an equation. It may still be illuminating to look at the numerical values of the roots returned by the polynomial root finder **allroots**, which we do for the above example:

```
(%i4) fpprintprec:8$
(%i5) nsolns : allroots(x^6 - 1);
(%o5) [x = 0.866025 %i + 0.5, x = 0.5 - 0.866025 %i, x = 0.866025 %i - 0.5,
                                   x = - 0.866025 %i - 0.5, x = 1.0, x = - 1.0]
(%i6) for i thru length(nsolns) do
        disp ( ev( x^6 - 1, nsolns[i],expand ))$
                      - 4.4408921E-16 %i - 5.55111512E-16
                       4.4408921E-16 %i - 5.55111512E-16
                       4.4408921E-16 %i - 5.55111512E-16
                      - 4.4408921E-16 %i - 5.55111512E-16
                                      0.0
                                      0.0
```

The numerical roots found by **allroots** satisfy the starting equation to within floating point errors.

Example 2

You can use **solve** to look for solutions to a set of equations. Here we find four sets of solutions satisfying two simultaneous equations, and we check the solutions.

```
(%i1) fpprintprec:8$
(%i2) eqns : [4*x^2 - y^2 - 12 = 0, x*y - x - 2 = 0]$
(%i3) solns : solve(eqns,[x,y]);
(%o3) [[x = 2, y = 2], [x = 0.520259 %i - 0.133124,
y = 0.0767838 - 3.6080032 %i], [x = - 0.520259 %i - 0.133124,
y = 3.6080032 %i + 0.0767838], [x = - 1.7337518, y = - 0.153568]]
(%i4) eqns,solns[1],ratsimp;
(%o4)                          [0 = 0, 0 = 0]
(%i5) for i:2 thru 4 do
        disp ( ev (eqns, solns[i],expand) )$
        [1.77635684E-15 - 6.66133815E-16 %i = 0, - 2.22044605E-16 = 0]
        [6.66133815E-16 %i + 1.77635684E-15 = 0, - 2.22044605E-16 = 0]
                  [- 1.13954405E-6 = 0, - 9.38499825E-8 = 0]
(%i6) solns[4];
(%o6)                  [x = - 1.7337518, y = - 0.153568]
```

We only use **ratsimp** on the first (integral) solution, which is an exact solution. For the numerical solutions, we use instead **expand**. The second and third (numerical) solutions have an accuracy of about 15 digits. The fourth (numerical) solution has about five digit accuracy.

Example 3: realroots

A Maxima user asked (on the Mailing List) how to find the value of **r** implied by the equation

```
275.0 * exp(-r) + 275.0 * exp(-2*r) + 275.0 * exp(-3*r)
  + 275.0 * exp(-4*r) + 5275.0 * exp(-5*r) = 4750.0
```

Straightforward use of **solve** does not return an explicit solution. Here is an approach suggested by Maxima developer Stavros Macrakis.

First use **rat** to convert approximate numbers (275.0) to exact (275), replace **exp(r)** by **z**, and then try **solve**.

```
(%i1) fpprintprec:8$
(%i2) eqn : 275.0 * exp(-r) + 275.0 * exp(-2*r) + 275.0 * exp(-3*r)
    + 275.0 * exp(-4*r) + 5275.0 * exp(-5*r) - 4750.0 = 0$
(%i3) rat(eqn),ratprint:false;
(%o3)/R/
                r 5          r 4          r 3          r 2         r
        4750 (%e )  - 275 (%e )  - 275 (%e )  - 275 (%e )  - 275 %e  - 5275
      - -------------------------------------------------------------------- = 0
                                          r 5
                                      (%e )
(%i4) zeqn : ratsubst(z,exp(r),%);
                    5        4        3        2
              4750 z  - 275 z  - 275 z  - 275 z  - 275 z - 5275
(%o4)       - ------------------------------------------------ = 0
                                      5
                                     z
(%i5) soln : solve(zeqn);
                          5       4       3       2
(%o5)           [0 = 190 z  - 11 z  - 11 z  - 11 z  - 11 z - 211]
```

We see that **solve** returned a list containing one element: a simplified form of our polynomial (in **z**) equation, but was unable to return an algebraic solution. We must then consider an approximate solution, and suppose we are only interested in real roots **z**, in which case we can use **realroots**.

```
(%i6) rr : realroots(soln[1]);
                                    35805935
(%o6)                          [z = --------]
                                    33554432
(%i7) zeqn, rr, ratsimp;
                    1011365300358912057317883590855138
(%o7)             - ------------------------------------ = 0
                    2354155174380926220896751357249338375
(%i8) %, numer;
(%o8)                       - 4.29608596E-4 = 0
```

We see that the returned real root is not very accurate, so we override the default value of **rootsepsilon** (which is one part in 10^7)

```
(%i9) rr : realroots( soln[1],1.0e-16);
                                38446329182573765
(%o9)                      [z = -----------------]
                                36028797018963968
(%i10) zeqn, rr, numer;
(%o10)                      - 6.5731657E-13 = 0
```

If $z = e^r$, then $\ln(z) = \ln(e^r) = r$. We then get an accurate numerical answer for `r` via

```
(%i11) rval : log(rhs(rr[1])),numer;
(%o11)                             0.0649447
(%i12) eqn, r = rval;
(%o12)                     - 1.70530257E-12 = 0
```

Example 4: Using find_root

Let's go thru the first manual example for **find_root**, which has the syntax

```
    find_root ( expr, var, a, b )
```

which assumes **expr** is a function of **var**, and Maxima is to look for a numerical root of the equation `expr = 0` in the range `a <= var <= b`.

Here we find the value of `x` such that `sin(x) - x/2 = 0`. The function **solve** cannot cope with this type of nonlinear problem.

```
(%i1) e : sin(x) - x/2;
                                          x
(%o1)                            sin(x) - -
                                          2
(%i2) solve(e);
(%o2)                          [x = 2 sin(x)]
```

Let's use **plot2d** for a quick look at this expression for positive **x**.

```
(%i3) plot2d(e,[x,0,2*%pi],
          [style,[lines,5]],[ylabel," "],
           [xlabel," sin(x) - x/2 " ],
           [gnuplot_preamble,"set zeroaxis lw 2"])$
```

which produces the plot:

Figure 1: $\sin(x) - x/2$

We see from the plot that there is a root in the range `0 < x < %pi` . We can now ask **find_root** to locate this root numerically.

```
(%i4) xr : find_root(e,x,0.1,%pi);
(%o4)                        1.895494267033981
(%i5) e, x = xr;
(%o5)                               0.0
```

Example 5: Helping solve Find the Exact Roots

Let's find the values of **x** which satisfy the equation

$$\mathbf{a}\,(\mathbf{1} - \sin(\mathbf{x})) = \mathbf{2}\,\mathbf{b}\cos(\mathbf{x}) \tag{1.1}$$

As this equation stands, **solve** is not able to make progress.

```
(%i1) eqn : a*(1-sin(x)) - 2*b*cos(x) = 0$
(%i2) solve ( eqn, x );
                                        2 b cos(x) - a
(%o2)                         [sin(x) = - --------------]
                                              a
```

There are several ways one can help out **solve** to get the solutions we want.

Method 1 is to convert the trig functions to their complex exponential forms using **exponentialize**.

```
(%i3) eqn, exponentialize;
                   %i x     - %i x
             %i (%e     - %e      )               %i x     - %i x
(%o3)     a (---------------------- + 1) - b (%e     + %e      ) = 0
                        2
(%i4) solns : solve ( %, x );
                        2 %i b         a
(%o4) [x = - %i log(---------- + ----------),
                    2 b - %i a   2 b - %i a
                                                             a           2 %i b
                                      x = - %i log(---------- - ----------)]
                                                   2 b - %i a   2 b - %i a
(%i5) solns : solns, rectform, ratsimp;
                                                 2    2
                      %pi               4 b  - a      4 a b
(%o5)            [x = ---, x = - atan2(---------, ---------)]
                       2                  2    2     2    2
                                       4 b  + a   4 b  + a
(%i6) eqn, solns[1], ratsimp;
(%o6)                              0 = 0
(%i7) eqn, solns[2], ratsimp;
(%o7)                              0 = 0
(%i8) x2 : atan2(a^2 - 4*b^2, 4*a*b)$
(%i9) eqn, x = x2, ratsimp;
                           4      2  2    4          2    3
               a sqrt(16 b  + 8 a  b  + a ) - 4 a b  - a
(%o9)        - ----------------------------------------- = 0
                                 2    2
                              4 b  + a
(%i10) scanmap( 'factor, % );
(%o10)                             0 = 0
```

Method 2 is to enlarge the system of equations to include the trig identity

$$\cos^2 x + \sin^2 x = 1 \tag{1.2}$$

which one would hope would be used creatively by **solve** to find the solutions of our equation.

Let's replace **cos(x)** by the symbol **c** and likewise replace **sin(x)** by the symbol **s**, and start with a pair of equations.

```
(%i1) eqns : [a*(1-s) - 2*b*c = 0, c^2 + s^2 - 1 = 0]$
(%i2) solns : solve ( eqns, [s,c] );
                                              2    2
                                           4 b  - a          4 a b
(%o2)        [[s = 1, c = 0], [s = - ---------, c = ---------]]
                                              2    2           2    2
                                           4 b  + a       4 b  + a
(%i3) eqns, solns[1], ratsimp;
(%o3)                         [0 = 0, 0 = 0]
(%i4) x1soln : solve ( sin(x) = 1, x );
solve: using arc-trig functions to get a solution.
Some solutions will be lost.
                                              %pi
(%o4)                              [x = ---]
                                               2
(%i5) eqn, x1soln, ratsimp;
(%o5)                            0 = 0
```

```
(%i6) eqns, solns[2], ratsimp;
(%o6)                        [0 = 0, 0 = 0]
(%i7) tan(x) = s/c, solns[2], ratsimp;
(%o7) tan(x) = -(4*b^2-a^2)/(4*a*b)
(%i8) x2soln : solve (%, x), ratsimp;
solve: using arc-trig functions to get a solution.
Some solutions will be lost.
                                          2    2
                                       4 b  - a
(%o8)                      [x = - atan(---------)]
                                         4 a b
```

Bearing in mind that $\arctan(-x) = -\arctan(x)$, we get the same set of solutions using method 2.

1.8.13 Non-Rational Simplification: radcan, logcontract, rootscontract, and radexpand

The wxMaxima interface has the **Simplify(r)** button allows uses the **radcan** function on a selection. The “r” stands for **radcan** although you could just as well think ”radical”, since it is equivalent to the wxMaxima menu choice **Simplify, Simplify Radicals**. You can also think of the work **radcan** as a mnenomic constructed from the capitalised letters in “RADical CANcellation”, but this does not hint at the full power of this function.

The manual asserts that **radcan** is useful in simplifying expressions containing logs, exponentials, and radicals. Here is an expression containing exponentials which **radcan** simplifies.

```
(%i1) expr : (exp(x)-1)/(exp(x/2)+1);
                                       x
                                     %e  - 1
(%o1)                               ---------
                                      x/2
                                    %e    + 1
(%i2) expr, ratsimp;
                                       x
                                     %e  - 1
(%o2)                               ---------
                                      x/2
                                    %e    + 1
(%i3) expr, radcan;
                                      x/2
(%o3)                               %e    - 1
```

We see that **radcan** is able to simplify this expression containing exponentials, whereas **ratsimp** cannot.

Here is some Xmaxima work showing the differences between the use of **ratsimp**, **radcan**, and **logcontract** on an expression which is a sum of logs.

```
(%i4) logexpr : log ( (x+2)*(x-2) ) + log(x);
(%o4)                    log((x - 2) (x + 2)) + log(x)
(%i5) logexpr, ratsimp;
                                      2
(%o5)                        log(x  - 4) + log(x)
(%i6) logexpr, fullratsimp;
                                      2
(%o6)                        log(x  - 4) + log(x)
(%i7) logexpr, radcan;
(%o7)                  log(x + 2) + log(x) + log(x - 2)
(%i8) %, logcontract;
                                          3
(%o8)                            log(x  - 4 x)
```

```
(%i9) logexpr, logcontract;
                                    3
(%o9)                          log(x  - 4 x)
```

We see that use of **logcontract** provided the same quality of simplification as the two step route **radcan** followed by **logcontract**.

Here we expolore the use of **radcan** and **rootscontract** to simplify square roots:

```
(%i10) sqrt(2)*sqrt(3), radcan;
(%o10)                         sqrt(2) sqrt(3)
(%i11) sqrt(2)*sqrt(3), rootscontract;
(%o11)                             sqrt(6)
(%i12) sqrt(6)*sqrt(3), radcan;
(%o12)                           3 sqrt(2)
(%i13) sqrt(6)*sqrt(3), rootscontract;
(%o13)                           3 sqrt(2)
(%i14) sqrt(6)/sqrt(3), radcan;
(%o14)                             sqrt(2)
(%i15) sqrt(6)/sqrt(3), rootscontract;
(%o15)                             sqrt(2)
```

radexpand

The Maxima manual describes the option variable **radexpand** which can have the values **true** (the default), **all**, or **false**. The setting of **radexpand** controls the automatic simplifications of radicals.

In the default case (**radexpand** set to **true**), `sqrt(x^2)` simplifies to `abs(x)`. This is because, by default, all symbols, in the absence of special declarations, are considered to represent real numbers. You can declare that all symbols should (unless declared otherwise) be considered to represent complex numbers using the **domain** flag, which has the default value **real**.

```
(%i16) sqrt(x^2);
(%o16)                             abs(x)
(%i17) domain:complex$
(%i18) sqrt(x^2);
                                        2
(%o18)                           sqrt(x )
(%i19) domain:real$
(%i20) sqrt(x^2);
(%o20)                             abs(x)
```

Next we show the simplifications which occur to the expression `sqrt(16*x^2)` for the three possible values of **radexpand**.

```
(%i21) radexpand;
(%o21)                              true
(%i22) sqrt(16*x^2);
(%o22)                           4 abs(x)
(%i23) radexpand:all$
(%i24) sqrt(16*x^2);
(%o24)                               4 x
(%i25) radexpand:false$
(%i26) sqrt(16*x^2);
                                          2
(%o26)                          sqrt(16 x )
```

1.8.14 Trigonometric Simplification: trigsimp, trigexpand, trigreduce, and trigrat

Here are simple examples of the important trig simplification functions **trigsimp**, **trigexpand**, **trigreduce**, and **trigrat**.

trigsimp converts trig functions to sines and cosines and also attempts to use the identiy $\cos(x)^2 + \sin(x)^2 = 1$.

```
(%i1) trigsimp(tan(x));
                                   sin(x)
(%o1)                              ------
                                   cos(x)
(%i2) sin(x+y), trigexpand;
(%o2)                   cos(x) sin(y) + sin(x) cos(y)
(%i3) x+3*cos(x)^2 - sin(x)^2, trigreduce;
                     cos(2 x)      cos(2 x)    1         1
(%o3)                -------- + 3 (-------- + -) + x - -
                        2             2        2         2
(%i4) trigrat ( sin(3*a)/sin(a+%pi/3) );
(%o4)                  sqrt(3) sin(2 a) + cos(2 a) - 1
```

The manual entry for **trigsimp** advises that **trigreduce**, **ratsimp**, and **radcan** may be able to further simplify the result. The author has the following function defined in **mbe1util.mac**, which is loaded in during startup.

```
    rtt(e) := radcan ( trigrat ( trigsimp (e)))$
```

This function can be equivalent to **trigreduce** if only sines and cosines are present, but otherwise may return different forms, as we show here, comparing the opposide effects of **trigexpand** and **trigreduce**. Note that **trigexpand** expands trig functions of sums of angles and/or trig functions of multiple angles (like `2*x` at the "top level", and the user should read the manual description to see options available.

```
(%i5) e: sin(x+y), trigexpand;
(%o5)                   cos(x) sin(y) + sin(x) cos(y)
(%i6) e, trigreduce;
(%o6)                            sin(y + x)
(%i7) rtt (e);
(%o7)                            sin(y + x)
(%i8) e :  tan(x+y), trigexpand;
                               tan(y) + tan(x)
(%o8)                         -----------------
                              1 - tan(x) tan(y)
(%i9) e, trigreduce;
                          tan(y)              tan(x)
(%o9)                - ----------------- - -----------------
                       tan(x) tan(y) - 1   tan(x) tan(y) - 1
(%i10) %, ratsimp;
                                tan(y) + tan(x)
(%o10)                        - -----------------
                                tan(x) tan(y) - 1
(%i11) rtt (e);
                                 sin(y + x)
(%o11)                           ----------
                                 cos(y + x)
(%i12) trigsimp( tan(x+y) );
                                 sin(y + x)
(%o12)                           ----------
                                 cos(y + x)
(%i13) e :  cosh(x + y), trigexpand;
(%o13)                 sinh(x) sinh(y) + cosh(x) cosh(y)
(%i14) e, trigreduce;
(%o14)                           cosh(y + x)
```

```
(%i15) rtt (e);
                                  - y - x    2 y + 2 x
                                %e        (%e          + 1)
(%o15)                          ---------------------------
                                             2
(%i16) expand(%);
                                    y + x     - y - x
                                  %e        %e
(%o16)                            ------- + ---------
                                     2          2
```

In the following, note the difference when using `trigexpand(expr)` compared with `ev ( expr, trigexpand )`. This difference is due to the fact that **trigexpand** has both the **evfun** and the **evflag** properties.

```
(%i1) e1 : trigexpand( tan(2*x + y) );
                                   tan(y) + tan(2 x)
(%o1)                             -------------------
                                  1 - tan(2 x) tan(y)
(%i2) e2 : tan (2*x + y), trigexpand;
                                         2 tan(x)
                                tan(y) + -----------
                                                2
                                         1 - tan (x)
(%o2)                           --------------------
                                    2 tan(x) tan(y)
                                1 - ---------------
                                              2
                                      1 - tan (x)
(%i3) rtt (e1);
                                    sin(y + 2 x)
(%o3)                               ------------
                                    cos(y + 2 x)
(%i4) rtt (e2);
                                    sin(y + 2 x)
(%o4)                               ------------
                                    cos(y + 2 x)
(%i5) trigsimp (e1);
                           cos(2 x) sin(y) + sin(2 x) cos(y)
(%o5)                    - ---------------------------------
                           sin(2 x) sin(y) - cos(2 x) cos(y)
(%i6) trigsimp (e2);
                     2
              (2 cos (x) - 1) sin(y) + 2 cos(x) sin(x) cos(y)
(%o6)       - -----------------------------------------------
                                                   2
              2 cos(x) sin(x) sin(y) + (1 - 2 cos (x)) cos(y)
```

1.8.15 Complex Expressions: rectform, demoivre, realpart, imagpart, and exponentialize

Sec. 6.2 of the Maxima manual has the introduction

> A complex expression is specified in Maxima by adding the real part of the expression to `%i` times the imaginary part. Thus the roots of the equation `x^2 - 4*x + 13 = 0` are `2 + 3*%i` and `2 - 3*%i`. Note that simplification of products of complex expressions can be effected by expanding the product. Simplification of quotients, roots, and other functions of complex expressions can usually be accomplished by using the **realpart**, **imagpart**, **rectform**, **polarform**, **abs**, **carg** functions.

Here is a brief look at some of these functions, starting with **exponentialize**.

```
(%i1) cform : [cos(x),sin(x),cosh(x)], exponentialize;
             %i x     - %i x          %i x     - %i x      x     - x
           %e     + %e            %i (%e     - %e      )  %e  + %e
(%o1)     [-----------------, - ----------------------, -----------]
                  2                       2                    2
(%i2) cform, demoivre;
                                                x     - x
                                              %e  + %e
(%o2)                      [cos(x), sin(x), -----------]
                                                 2
(%i3) cform, rectform;
                                                x     - x
                                              %e  + %e
(%o3)                      [cos(x), sin(x), -----------]
                                                 2
(%i4) realpart ( cform );
                                                x     - x
                                              %e  + %e
(%o4)                      [cos(x), sin(x), -----------]
                                                 2
(%i5) imagpart ( cform );
(%o5)                                 [0, 0, 0]
```

1.8.16 Are Two Expressions Numerically Equivalent? zeroequiv

The function **zeroequiv(expr, var)** uses a series of randomly chosen values of the variable **var** to test if **expr** is equivalent to zero. Although there are cases where **dontknow** is returned (see the manual cautions), this function can be useful if you arrange that **expr** is the difference of two other expressions which you suspect are numerically equivalent (or not).

As a pedagogical example consider showing that

$$\cos^2(x-1) = \frac{1}{2}\left(\sin(2)\ \sin(2\,x) + \cos(2)\ \cos(2x) + 1\right) \tag{1.3}$$

is true. Although it is not difficult to show the equivalence using the standard Maxima tools (see Sec. 10.3.6 in Ch.10), we use here **zeroequiv**.

```
(%i1) e1:cos(x-1)^2;
                                        2
(%o1)                                cos  (x - 1)
(%i2) e2 : (sin(2)*sin(2*x)+cos(2)*cos(2*x)+1)/2;
                      sin(2) sin(2 x) + cos(2) cos(2 x) + 1
(%o2)                 -------------------------------------
                                        2
(%i3) zeroequiv ( e1-e2, x );
(%o3)                                   true
```

An alternative (and complementary) approach is to plot both expressions `e1` and `e2` on the same plot.

1.9 User Defined Maxima Functions: define, fundef, block, and local

A Maxima function can be defined using the `:=` operator. The left side of the function definition should be the name of the function followed by comma separated formal parameters enclosed in parentheses. The right side of the Maxima function definition is the "function body". When a Maxima function is called, the formal parameters are bound to the call arguments, any "free" variables in the function body take on the values that they have at the time of the function call, and the function body is evaluated. You can define Maxima functions which are recursive to an "arbitrary" depth (of course there are always practical limits due to memory, speed,...). After a Maxima function is defined, its name is added to the Maxima list **functions**.

Problems may sometimes arise when passing to a Maxima function (as one of the calling arguments) an expression which contains a variable with the same name as a formal parameter used in the function body (name conflicts).

1.9.1 A Function Which Takes a Derivative

In this section we discuss a question sent in to the Maxima Mailing List (which we paraphrase).

```
I want to make the derivative of a function a function itself.
Naively, I tried the following.
First define a Maxima function f(x) as a simple polynomial.
Next define a second Maxima function fp(x) ("f-prime")
  which will take the first derivative of f(x).
Then try to use fp(x) to get the derivative of f at x =1.

This did not work, as shown here:
------------------------------------------------
(%i1) f(x):= -2*x^3 + 3*x^2 + 12*x - 13$
(%i2) fp(x):= diff(f(x),x)$
(%i3) fp(1);
diff: second argument must be a variable; found 1
#0: fp(x=1)
 -- an error.  To debug this try debugmode(true);
---------------------------------------------
What can I do?
Thanks
```

The error message returned by Maxima in response to input `%i3` means that Maxima determined that the job requested was to return the result of the operation `diff( f(1), 1 )`. In other words, "variable substitution" occurred before differentiation, whereas, what is wanted is first differentiation, and then variable substitution.

Let's pick a simpler function, and compare three ways to try to get this job done.

```
(%i1) f(y) := y^3;
                                                   3
(%o1)                             f(y) := y
(%i2) f(z);
                                               3
(%o2)                                         z
(%i3) f1(x) := diff ( f(x), x);
(%o3)                       f1(x) := diff(f(x), x)
(%i4) f2(x) := ''(diff ( f(x), x));
                                                     2
(%o4)                            f2(x) := 3 x
(%i5) define ( f3(x), diff ( f(x), x) );
                                                     2
(%o5)                            f3(x) := 3 x
```

```
(%i6) [f1(z), f2(z), f3(z) ];
                                     2     2     2
(%o6)                           [3 z , 3 z , 3 z ]
(%i7) [f1(1), f2(1), f3(1) ];
diff: second argument must be a variable; found 1
#0: f1(x=1)
 -- an error.  To debug this try debugmode(true);
(%i8) [f2(1), f3(1)];
(%o8)                                  [3, 3]
```

As long as we request the derivative of f(z), where z is treated as an undefined variable (ie., **z** has not been bound to a number, in which case `numberp(z)` will return `false`), all three definitions give the correct result. But when we request the derivative at a specific numerical point, only the last two methods give the correct result without errors, and the first (”delayed assignment operator method”) does not work.

Note that the “quote-quote method” requires two single quotes, and we need to surround the whole expression with parentheses: `''(...)`. This “quote-quote” method is available only in interactive use, and will not work inside another function, as you can verify. The conventional wisdom denigrates the quote-quote method and advises use of the **define** method.

The **define** function method can not only be used interactively, but also when one needs to define a function like this inside another Maxima function, as in

```
(%i9) dplay(g,a,b) := block ( [val1,val2 ],
        local (dg),
          define ( dg(y), diff (g(y), y)),
          val1 : g(a) + dg(a),
          val2 : g(b) + dg(b),
          display (val1,val2),
          val1 + val2)$
(%i10) g(x) := x;
(%o10)                          g(x) := x
(%i11) dplay(g,1,1);
                                val1 = 2

                                val2 = 2

(%o11)                              4
(%i12) g(x) := x^2;
                                        2
(%o12)                         g(x) := x
(%i13) dplay(g,1,2);
                                val1 = 3

                                val2 = 8

(%o13)                             11
(%i14) dg(3);
(%o14)                            dg(3)
```

You can find the manual discussion of the quote-quote operator near the top of the index list where misc. symbols are recorded. There you find the comment “Applied to the operator of an expression, quote-quote changes the operator from a noun to a verb (if it is not already a verb).”

The use of **local** inside our Maxima function **dplay** prevents our definition of the Maxima function **dg** from leaking out into the top level context and keeps the knowledge of **dg** inside **dplay** and any user defined Maxima functions which are called by **dplay**, and inside any user defined Maxima functions which are called by those functions, etc...

Maxima developer Stavros Macrakis has posted a good example which emphasizes the difference between the delayed assignment operator **:=** and the **define** function. This example uses Maxima's **print** function.

```
(%i1) f1(x):=print (x)$
(%i2) define ( f2(x), print (x) )$
x
```

Notice that the delayed assignment definition for **f1** did not result in the actual operation of **print(x)**, whereas the **define** definition of **f2** resulted in the operation of **print(x)**, as you can see on your screen after input **%i2**. Quoting from Macrakis:

> In the case of **':='**, both arguments are unevaluated. This is the normal way function definitions work in most programming languages.
>
> In the case of **'define'**, the second argument is evaluated normally [in the process of absorbing the definition], not unevaluated. This allows you to ***calculate*** a definition.

We can see the difference between **f1** and **f2** by using **fundef**, which returns what Maxima has accepted as a definition in the two cases.

```
(%i3) fundef (f1);
(%o3)                          f1(x) := print(x)
(%i4) fundef (f2);
(%o4)                             f2(x) := x
```

Let's now compare these definitions at runtime. **f1** will print its argument to the screen, but **f2** will not, since in the latter case, all reference to Maxima's **print** function has been lost.

```
(%i5) f1(5);
5
(%o5)                                   5
(%i6) f2(5);
(%o6)                                   5
```

Macrakis continues:

> **'define'** is useful for calculating function definitions which will later be applied to arguments. For example, compare:

```
(%i7) define (f(x), optimize (horner (taylor (tanh(x),x,0,8),x)));
                                 2  x (%1 (%1 (42 - 17 %1) - 105) + 315)
(%o7) f(x) := block([%1], %1 : x , ------------------------------------)
                                                  315
(%i8) f(0.1);
(%o8)                         0.099667994603175
(%i9) g(x) := optimize (horner (taylor (tanh(x),x,0,8),x));
(%o9)        g(x) := optimize(horner(taylor(tanh(x), x, 0, 8), x))
(%i10) g(0.1);
Variable of expansion cannot be a number: 0.1
#0: g(x=0.1)
 -- an error.  To debug this try debugmode(true);
```

g(.1) => ERROR (because it is trying to do **taylor(tanh(.1),.1,0,8)**).

1.9.2 Lambda Expressions

Lambda notation is used to define unnamed "anonymous" Maxima functions. Lambda expressions allow you to define a special purpose function which will not be used in other contexts (since the function has no name, it cannot be used later).

The general syntax is
lambda (arglist, expr1, expr2, ...,exprn)
where arglist has the form **[x1, x2, ...,xm]** and provide formal parameters in terms of which the **expr1, expr2,...** can be written, (use can also be made of %% to refer to the result of the previous expression **exprj**). When this function is evaluated, unbound local variables **x1,x2,...** are created, and then **expr1** through **exprn** are evaluated in turn. The return value of this function is **exprn**.

```
(%i1) functions;
(%o1) [qplot(exprlist, prange, [hvrange]), rtt(e), ts(e, v), to_atan(e, y, x),
to_atan2(e, y, x), totan(e, v), totan2(e, v), mstate(), mattrib(), mclean(),
fll(x)]
(%i2) f : lambda([x,y],x^2 + y^3);
                                                  2    3
(%o2)                         lambda([x, y], x  + y )
(%i3) f (1,1);
(%o3)                                        2
(%i4) f(2,a);
                                           3
(%o4)                                     a  + 4
(%i5) apply ( f, [1,2] );
(%o5)                                      9
(%i6) functions;
(%o6) [qplot(exprlist, prange, [hvrange]), rtt(e), ts(e, v), to_atan(e, y, x),
to_atan2(e, y, x), totan(e, v), totan2(e, v), mstate(), mattrib(), mclean(),
fll(x)]
(%i7) map ( 'sin, [1,2,3] );
(%o7)                        [sin(1), sin(2), sin(3)]
(%i8) map ( lambda([x],x^2 + sin(x) ),[1,2,3] );
(%o8)                  [sin(1) + 1, sin(2) + 4, sin(3) + 9]
(%i9) lambda ([x],x+1 )(3);
(%o9)                                      4
```

1.9.3 Recursive Functions; factorial, and trace

Here is an example which does the same jub as the Maxima function **factorial**. We turn on **trace** to watch the recursion process.

```
(%i1) myfac(n) := if n = 0 then 1 else n*myfac(n-1)$
(%i2) map ('myfac, [0,1,2,3,4] );
(%o2)                           [1, 1, 2, 6, 24]
(%i3) [0!,1!,2!,3!,4!];
(%o3)                           [1, 1, 2, 6, 24]
(%i4) map ('factorial, [0,1,2,3,4] );
(%o4)                           [1, 1, 2, 6, 24]
(%i5) trace (myfac);
(%o5)                                [myfac]
```

```
(%i6) myfac(4);
1 Enter myfac [4]
 2 Enter myfac [3]
  3 Enter myfac [2]
   4 Enter myfac [1]
    5 Enter myfac [0]
    5 Exit  myfac 1
   4 Exit  myfac 1
  3 Exit  myfac 2
 2 Exit  myfac 6
1 Exit  myfac 24
(%o6)                                  24
(%i7) untrace(myfac);
(%o7)                                [myfac]
(%i8) [myfac(4),factorial(4),4!];
(%o8)                             [24, 24, 24]
```

Legendre Polynomial

As a second example of a recursive definition of a Maxima function, one can define the Legendre polynomial $P_n(x)$ via a recursive relation that relates three different **n** values (for a given **x**). We can check our design by using the Maxima function **legendre_p (n,x)**, which is part of the orthogonal polynomial package **orthopoly.lisp**.

```
(%i1) p(n,x) := if n=0 then 1 elseif n=1 then x else
             expand( ( (2*n-1)/n )*x*p(n-1,x) - ((n-1)/n)*p(n-2,x) )$
(%i2) [p(0,x),p(1,x),p(2,x),p(3,x)];
                                  2          3
                               3 x    1   5 x    3 x
(%o2)                    [1, x, ---- - -, ---- - ---]
                                2     2   2       2
(%i3) map ( lambda([nn], p (nn,x) ),[0,1,2] );
                                        2
                                     3 x    1
(%o3)                          [1, x, ---- - -]
                                      2     2
(%i4) map ( lambda([nn], expand (legendre_p (nn,x))),[0,1,2] );
                                        2
                                     3 x    1
(%o4)                          [1, x, ---- - -]
                                      2     2
```

1.9.4 Non-Recursive Subscripted Functions (Hashed Arrays)

"Undeclared arrays" are called "hashed arrays" and they grow dynamically as more "array elements" are assigned values or otherwise defined. Here is a non-recursive simple subscripted function having one subscript **k** and one formal argument **x**.

```
(%i1) f[k](x) := x^k + 1;
                                          k
(%o1)                          f (x) := x  + 1
                                k
(%i2) arrays;
(%o2)                                 [f]
(%i3) makelist ( f[nn](y),nn,0,3 );
                                     2       3
(%o3)                     [2, y + 1, y  + 1, y  + 1]
(%i4) arrayinfo(f);
(%o4)                   [hashed, 1, [0], [1], [2], [3]]
```

```
(%i5) map ( lambda ( [m], f[m](y) ), [0,1,2,3] );
                                          2        3
(%o5)                      [2, y + 1, y  + 1, y  + 1]
```

Here is a example which uses Maxima's anonymous **lambda** function to define a subscripted function which does the same job as above.

```
(%i1) h[nn] := lambda ([xx], xx^nn + 1);
                                                   nn
(%o1)                     h    := lambda([xx], xx   + 1)
                           nn
(%i2) arrays;
(%o2)                                   [f, h]
(%i3) arrayinfo(h);
(%o3)                               [hashed, 1]
(%i4) h[2];
                                                2
(%o4)                        lambda([xx], xx  + 1)
(%i5) arrayinfo(h);
(%o5)                           [hashed, 1, [2]]
(%i6) h[2](y);
                                         2
(%o6)                                   y  + 1
(%i7) map ( lambda ( [mm], h[mm](x) ),[0,1,2,3] );
                                          2        3
(%o7)                      [2, x + 1, x  + 1, x  + 1]
(%i8) arrayinfo(h);
(%o8)                    [hashed, 1, [0], [1], [2], [3]]
```

1.9.5 Recursive Hashed Arrays and Memoizing

We again design a homemade factorial "function", this time using a "hashed array" having one "index" **n**.

```
(%i1) arrays;
(%o1)                                   []
(%i2) a[n] := n*a[n-1]$
(%i3) a[0] : 1$
(%i4) arrays;
(%o4)                                   [a]
(%i5) arrayinfo(a);
(%o5)                           [hashed, 1, [0]]
(%i6) a[4];
(%o6)                                   24
(%i7) arrayinfo(a);
(%o7)                 [hashed, 1, [0], [1], [2], [3], [4]]
(%i8) a[n] := n/2$
(%i9) a[4];
(%o9)                                   24
(%i10) a[7];
                                         7
(%o10)                                   -
                                         2
(%i11) arrayinfo(a);
(%o11)              [hashed, 1, [0], [1], [2], [3], [4], [7]]
```

We see in the above example that elements **1** through **4** were computed when the call **a[4]** was first made. Once the array elements have been computed, they are not recomputed, even though our definition changes. Note that after **(%i8) a[n] := n/2$**, our interogation **(%i9) a[4];** does not return our new definition, but the result of the original calculation. If you compute **a[50]** using our first recursive factorial definition, all values **a[1]** through **a[50]** are remembered. Computing **a[51]** then takes only one step.

1.9.6 Recursive Subscripted Maxima Functions

We here use a "subscripted function" to achieve a recursive definition of a Legendre polynomial $P_n(x)$. (The author thanks Maxima developer Richard Fateman for this example.)

```
(%i1) ( p[n](x) := expand(((2*n-1)/n)*x*p [n-1](x) - ((n-1)/n)*p [n-2](x) ),
        p [0](x) := 1, p [1](x) := x ) $
(%i2) makelist( p[m](x),m,0,4 );
                              2          3             4        2
                           3 x    1   5 x     3 x   35 x    15 x     3
(%o2)            [1, x, ---- - -, ---- - ---, ----- - ----- + -]
                             2     2   2      2      8       4     8
(%i3) arrays;
(%o3)                                 [p]
(%i4) arrayinfo(p);
(%o4)                  [hashed, 1, [0], [1], [2], [3], [4]]
(%i5) [p[3](x),p[3](y)];
                                    3           3
                                 5 x    3 x  5 y    3 y
(%o5)                          [---- - ---, ---- - ---]
                                  2      2    2      2
(%i6) map ( lambda([nn], p[nn](x)),[0,1,2,3] );
                                        2         3
                                     3 x    1  5 x    3 x
(%o6)                        [1, x, ---- - -, ---- - ---]
                                      2     2   2      2
```

Note that the input line **`%i1`** has the structure
`( job1, job2, job3 )$`. This is similar to using one input line to bind values to several variables.

```
(%i7) ( a:1, b:2/3, c : cos(4/3) )$
```

If you want a visual reminder of the bindings, then the **`[job1,job2,..]`** notation is more appropriate.

```
(%i8) [ a:1, b:2/3, c : cos(4/3) ];
                                         2      4
(%o8)                                [1, -, cos(-)]
                                         3      3
```

1.9.7 Floating Point Numbers from a Maxima Function

Let's use our recursive subscripted function definition of $P_n(x)$ and compare the "exact" value with the default floating point number when we look for $P_3(8/10)$. Let **`p3t`** hold the exact value **`2/25`**. Maxima's **float** function returns **`0.08`** from **`float(2/25)`**, but internally Maxima holds a binary representation with has a small error when converted to a decimal number. We can see the internally held decimal number by using **`?print`** which accesses the lisp function **print** from Maxima.

```
(%i1) ( p[n](x) := expand(((2*n-1)/n)*x*p [n-1](x) - ((n-1)/n)*p [n-2](x) ),
        p [0](x) := 1, p [1](x) := x ) $
(%i2) p3 : p[3](x);
                                          3
                                       5 x    3 x
(%o2)                                  ---- - ---
                                        2      2
(%i3) p3t : p3, x = 8/10;
                                          2
(%o3)                                     --
                                          25
```

```
(%i4) float(p3t);
(%o4)                              0.08
(%i5) ?print(%);
0.08000000000000002
(%o5)                              0.08
(%i6) p3f : p3, x = 0.8;
(%o6)                              0.08
(%i7) ?print(%);
0.08000000000000071
(%o7)                              0.08
(%i8) abs (p3f - p3t);
(%o8)                     6.938893903907228E-17
(%i9) abs (%o4 - p3t);
(%o9)                               0.0
(%i10) ?print(%);
0.0
(%o10)                              0.0
```

The input line **(%i6) p3f : p3, x = 0.8;** replaces **x** by the floating point decimal number **0.8** (which has no exact binary representation) and then uses binary arithmetic to perform the indicated additions, divisions, and multiplications, and then returns the closest decimal number corresponding to the final binary value. The result of all this hidden work is a number with some "floating point error", which we calculate in line **%i8**.

We can come to the same conclusion in a slightly different way if we first define a Maxima function **P3(x)** using **p[3]**, say.

```
(%i11) define (P3(y), p[3](y))$
(%i12) P3(x);
                                    3
                                 5 x    3 x
(%o12)                           ---- - ---
                                  2      2
(%i13) P3f1 : float (P3(8/10));
(%o13)                             0.08
(%i14) ?print(%);
0.08000000000000002
(%o14)                             0.08
(%i15) P3f2 : P3(0.8);
(%o15)                             0.08
(%i16) ?print(%);
0.08000000000000071
(%o16)                             0.08
(%i17) abs (P3f2 - P3f1);
(%o17)                    6.938893903907228E-17
```

We have used **?print(...)** above to look at the approximate decimal digit equivalent of a binary representation held internally by Maxima for a floating point number. We can sometimes learn new things by looking at Lisp translations of input and output using **?print(Maxima stuff)**, **:lisp #$ Maxima stuff $**, and **:lisp $_**. (See html Help Manual: Click on Right Panel, Click on Contents, Sec. 3.1, Lisp and Maxima.)

```
(%i18) ?print(a + b)$
((MPLUS SIMP) $A $B)
(%i19) :lisp $_
((PRINT) ((MPLUS) $A $B))
(%i19) :lisp #$a+b$
((MPLUS SIMP) $A $B)
(%i19) ?print( a/b )$
((MTIMES SIMP) $A ((MEXPT SIMP) $B -1))
(%i20) :lisp $_
((PRINT) ((MQUOTIENT) $A $B))
```

```
(%i20) :lisp #$a / b$
((MTIMES SIMP) $A ((MEXPT SIMP) $B -1))
(%i21) bfloat(2/25),fpprec:30;
(%o21)                            8.0b-2
(%i22) ?print(%);
((BIGFLOAT SIMP 102) 3245185536584267267831560205762 -3)
(%o22)                            8.0b-2
(%i23) p3, x = 8/10;
                                    2
(%o23)                              --
                                    25
(%i24) :lisp $_
(($EV) $P3 ((MEQUAL) $X ((MQUOTIENT) 8 10)))
(%i24) :lisp #$ p3, x=8/10 $
((RAT SIMP) 2 25)
```

1.10 Pulling Out Overall Factors from an Expression

If you have an expression of the form `expr : u * ( fac*c1 + fac*c2 )`, you can pull out the common factor `fac` using `fac * ratsimp ( expr / fac )` . Of course there may also be other methods which will work for a particular expression.

Here is one example.

```
(%i1) e1 : x*(A*cos(d1)^3 - A*cos(d2)^3);
                                 3           3
(%o1)                     x (cos  (d1) A - cos  (d2) A)
(%i2) A * ratsimp (e1/A);
                                3          3
(%o2)                      (cos  (d1) - cos  (d2)) x A
(%i3) rat (e1);
                                 3          3
(%o3)/R/                  (- cos  (d2) + cos  (d1)) x A
```

Here is a second example.

```
(%i4) e2 : x*(A*a*b1 - A*a*b2);
(%o4)                         x (a b1 A - a b2 A)
(%i5) A*a*ratsimp ( e2/( A*a));
(%o5)                           a (b1 - b2) x A
(%i6) factor(e2);
(%o6)                          - a (b2 - b1) x A
(%i7) rat(e2);
(%o7)/R/                       (- a b2 + a b1) x A
```

We see above that **rat** does not pull out all the common factors in this second example. We also find that **factor** does not simply pull out factors, but also rewrites the trig functions, when used with the first expression.

```
(%i8) factor (e1);
                                   2                           2
(%o8)  - (cos(d2) - cos(d1)) (cos  (d2) + cos(d1) cos(d2) + cos  (d1)) x A
```

More discussion of display and simplification tools will appear in the upcoming new Chapter 4.

1.11 Construction and Use of a Test Suite File

It is useful to construct a code test which can be run against new versions of your own personal code, as well as against new versions of Maxima. We show here a very simple example of such a code test file and use.

The test of floating point division (below) is sensitive to the specific Lisp version used to compile the Maxima source code; different Lisp versions may have slightly different floating point behaviors and digits of precision. Here are the contents of **mytest.mac**, which contains the tests: addition of integers, multiplication of integers, integer division, and floating point division. The convention is to include an input Maxima command followed by the expected Maxima answer (with the addition of a semicolon). The command `batch("mytest.mac",test)` will then use Maxima's test utilities. (You will find much useful and practical information in the file `README.how-to` in the `...\maxima\5.19.0\tests` folder.)

```
/* this is mytest.mac */
1 + 2;
3;
2 * 3;
6;
2/3;
2/3;
2.0/3;
0.66666666666666663;
```

Here is an example of running **mytest.mac** as a **batch** file in **test** mode using one dimensional output display.

```
(%i1) display2d:false$
(%i2) 2.0/3;
(%o2) 0.66666666666667
(%i3) ?print(%);
0.66666666666666663
(%o3) 0.66666666666667
(%i4) batch ("mytest.mac",test )$
Error log on #<output stream mytest.ERR>
********************** Problem 1 ***************
Input:
2+1
Result:
3
... Which was correct.

********************** Problem 2 ***************
Input:
2*3
Result:
6
... Which was correct.

********************** Problem 3 ***************
Input:
2/3
Result:
2/3
... Which was correct.

********************** Problem 4 ***************
Input:
2.0/3
Result:
0.66666666666667
... Which was correct.
4/4 tests passed.
```

Note that to avoid an error return on the floating point division test, we needed to use for the answer the result returned by the Lisp print function, which is available from Maxima by preceding the Lisp function name with a question mark (?).

1.12 History of Maxima's Development

From the Unix/Linux Maxima manual:

> MACSYMA (Project MAC's SYmbolic MAnipulation System) was developed by the Mathlab group of the MIT Laboratory for Computer Science (originally known as Project MAC), during the years 1969-1972. Their work was supported by grants NSG 1323 of the National Aeronautics and Space Administration, N00014-77-C-0641 of the Office of Naval Research, ET-78-C-02-4687 of the U.S. Department of Energy, and F49620-79-C-020 of the U.S. Air Force. MACSYMA was further modified for use under the UNIX operating system (for use on DEC VAX computers and Sun workstations), by Richard Fateman and colleagues at the University of California at Berkeley; this version of MACSYMA is known as VAXIMA. The present version stems from a re-working of the public domain MIT MACSYMA for GNU Common Lisp, prepared by William Schelter, University of Texas at Austin until his passing away in 2001. It contains numerous additions, extensions and enhancements of the original.

From the first page of the Maxima html help manual:

> Maxima is derived from the Macsyma system, developed at MIT in the years 1968 through 1982 as part of Project MAC. MIT turned over a copy of the Macsyma source code to the Department of Energy in 1982; that version is now known as DOE Macsyma. A copy of DOE Macsyma was maintained by Professor William F. Schelter of the University of Texas from 1982 until his death in 2001. In 1998, Schelter obtained permission from the Department of Energy to release the DOE Macsyma source code under the GNU Public License, and in 2000 he initiated the Maxima project at SourceForge to maintain and develop DOE Macsyma, now called Maxima.

Maxima is now developed and maintained by the Maxima project at
`http://maxima.sourceforge.net`.

Maxima by Example:
Ch. 2, Two Dimensional Plots and Least Squares Fits *

Edwin L. Woollett

September 16, 2010

Contents

2.1 Introduction to **plot2d** 3
  2.1.1 First Steps with **plot2d** 3
  2.1.2 **Parametric** Plots 6
  2.1.3 Line Width and Color Controls 9
  2.1.4 **Discrete** Data Plots: Point Size, Color, and Type Control 12
  2.1.5 More **gnuplot_preamble** Options 15
  2.1.6 Using **qplot** for Quick Plots of One or More Functions 16
2.2 Least Squares Fit to Experimental Data 18
  2.2.1 Maxima and Least Squares Fits: **lsquares_estimates** 18
  2.2.2 Syntax of **lsquares_estimates** 19
  2.2.3 Coffee Cooling Model 20
  2.2.4 Experiment Data: **file_search, printfile, read_nested_list,** and **makelist** 21
  2.2.5 Least Squares Fit of Coffee Cooling Data 22

*This version uses **Maxima 5.18.1.** Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions to `woollett@charter.net`

```
COPYING AND DISTRIBUTION POLICY
This document is part of a series of notes titled
"Maxima by Example" and is made available
via the author's webpage http://www.csulb.edu/~woollett/
to aid new users of the Maxima computer algebra system.

NON-PROFIT PRINTING AND DISTRIBUTION IS PERMITTED.

You may make copies of this document and distribute them
to others as long as you charge no more than the costs of printing.

These notes (with some modifications) will be published in book form
eventually via Lulu.com in an arrangement which will continue
to allow unlimited free download of the pdf files as well as the option
of ordering a low cost paperbound version of these notes.
```


Feedback from readers is the best way for this series of notes to become more helpful to new users of Maxima. *All* comments and suggestions for improvements will be appreciated and carefully considered.

```
LOADING FILES
The defaults allow you to use the brief version load(fft) to load in the
Maxima file fft.lisp.
To load in your own file, such as qxxx.mac
using the brief version load(qxxx), you either need to place
qxxx.mac in one of the folders Maxima searches by default, or
else put a line like:

file_search_maxima : append(["c:/work2/###.{mac,mc}"],file_search_maxima )$

in your personal startup file maxima-init.mac (see Ch. 1, Introduction to
  Maxima for more information about this).

Otherwise you need to provide a complete path in double quotes,
as in load("c:/work2/qxxx.mac"),

We always use the brief load version in our examples, which are generated
using the XMaxima graphics interface on a Windows XP computer, and copied
into a fancy verbatim environment in a latex file which uses the fancyvrb
and color packages.
```


```
Maxima.sourceforge.net. Maxima, a Computer Algebra System. Version 5.18.1
 (2009). http://maxima.sourceforge.net/
```

2.1 Introduction to plot2d

You should be able to use any of our examples with either **wxMaxima** or **Xmaxima**. If you substitute the word **wxplot2d** for the word **plot2d** you should get the same plot (using **wxMaxima**), but the plot will be drawn "inline" in your notebook rather than in a separate window.

To save a plot as an image file, using **wxMaxima**, right click on the inline plot, choose a name and a destination folder, and click ok.

To save a plot drawn in a separate **gnuplot** window, right click the small icon in the upper left hand corner of the plot window, choose Options, Copy to Clipboard, and then open any utility which can open a picture file and select Edit, Paste, and then File, Save As. A standard utility which comes with Windows XP is the accessory Paint, which will work fine in this role to save the clipboard image file. The freely available Infanview is a combination picture viewer and editor which can also be used for this purpose. Saving the image via the **gnuplot** window route results in a larger image.

2.1.1 First Steps with plot2d

The syntax of **plot2d** is

```
plot2d( object-list, draw-parameter-list, other-option-lists).
```

The required object list (the first item) may be simply one object (not a list). The object types may be expressions (or functions), all depending on the same draw parameter, discrete data objects, and parametric objects. If at least one of the plot objects involves a draw parameter, say **p**, then a draw parameter range list of the form **[p, pmin, pmax]** should follow the object list.

We start with the simplest version which only controls how much of the expression to plot, and does not try to control the canvas width or height.

```
(%i1) plot2d ( sin(u), [u, 0, %pi/2] )$
```

which produces the plot:

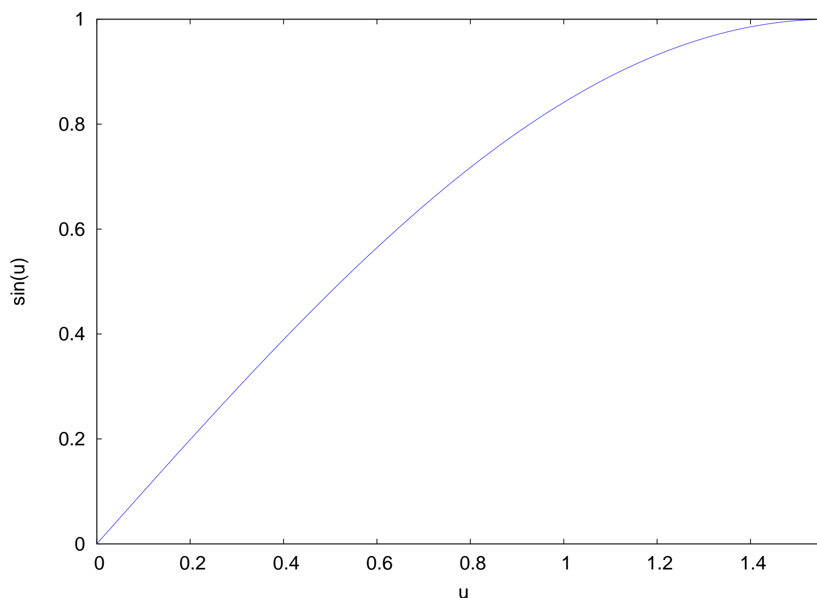

Figure 1: plot2d (sin(u), [u, 0, %pi/2])

We see that **plot2d** has made the canvas width only as wide as the drawing width, and has made the canvas height only as high as the drawing height. Now let's add a horizontal range (canvas width) control list in the form `[x,-0.2,1.8]`. Notice the special role the symbol `x` plays here in **plot2d**.

```
(%i2) plot2d ( sin(u),[u,0,%pi/2],[x, -0.2, 1.8] )$
```

which produces

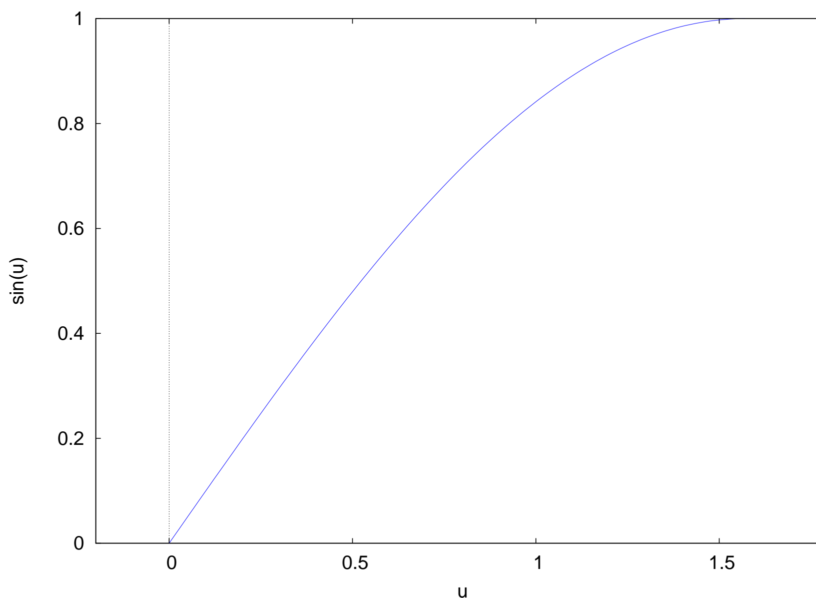

Figure 2: plot2d (sin(u), [u, 0, %pi/2], [x,-0.2,1.8])

We see that we now have separate draw width and canvas width controls included. If we try to put the canvas width control list before the draw width control list, we get an error message:

```
(%i3) plot2d(sin(u),[x,-0.2,1.8], [u,0,%pi/2] )$
Unknown plot option specified:  u
 -- an error.  To debug this try debugmode(true);
```

However, if the expression variable happens to be `x`, the following command includes both draw width and canvas width using separate `x` symbol control lists.

```
(%i4) plot2d ( sin(x), [x,0,%pi/2], [x,-0.2,1.8] )$
```

in which the first (required) `x` control list determines the drawing range, and the second (optional) `x` control list determines the canvas width.

Despite the special role the symbol `y` also plays in **plot2d**, the following command produces the same plot as above.

```
(%i5) plot2d ( sin(y), [y,0,%pi/2], [x,-0.2,1.8] )$
```

The optional vertical canvas height control list uses the special symbol **y**, as shown in

```
(%i6) plot2d ( sin(u), [u,0,%pi/2], [x,-0.2,1.8], [y,-0.2, 1.2] )$
```

which produces

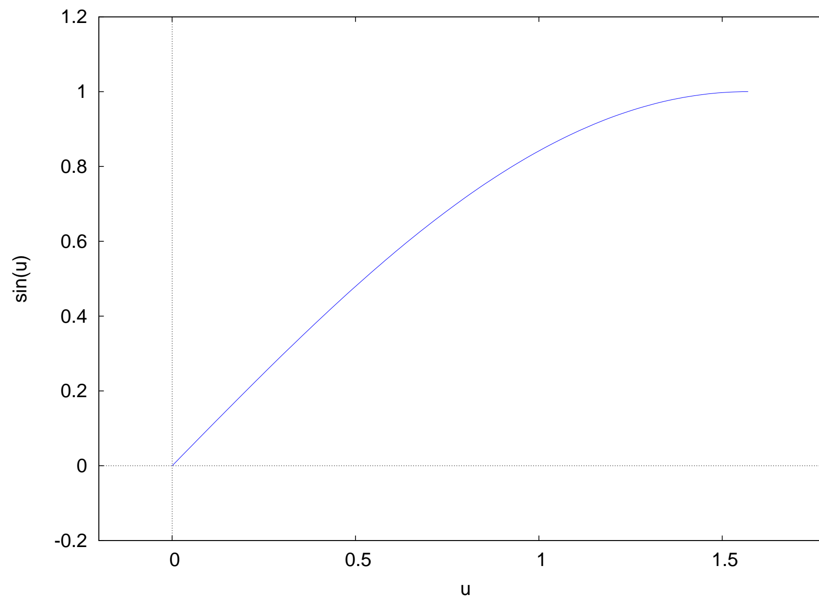

Figure 3: plot2d (sin(u), [u,0,%pi/2], [x,-0.2,1.8], [y,-0.2, 1.2])

and the following alternatives produce exactly the same plot.

```
(%i7) plot2d ( sin(u), [u,0,%pi/2], [y,-0.2, 1.2], [x,-0.2,1.8] )$
(%i8) plot2d ( sin(x), [x,0,%pi/2], [x,-0.2,1.8], [y,-0.2, 1.2] )$
(%i9) plot2d ( sin(y), [y,0,%pi/2], [x,-0.2,1.8], [y,-0.2, 1.2] )$
```

2.1.2 Parametric Plots

For orientation, we will draw a sine curve using the parametric plot object syntax and using a parametric parameter **t**.

```
(%i1) plot2d ( [parametric, t, sin(t), [t, 0, %pi] ] )$
```

which produces

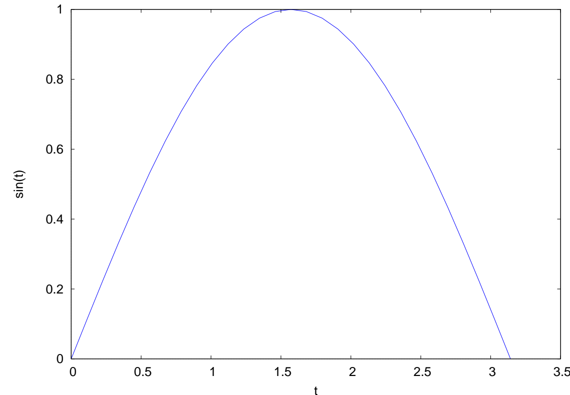

Figure 4: plot2d ([parametric, t, sin(t), [t, 0, %pi]])

We see that

```
plot2d ( [ parametric, fx(t), fy(t), [ t, tmin, tmax ] ] )$
```

plots pairs of points **(fx (ta), fy(ta))** for **ta** in the interval **[tmin, tmax]**. We have used no canvas width control list **[x, xmin, xmax]** in this minimal version.

We next use a parametric plot to create a "circle", letting **fx(t) = cos(t)** and **fy(t) = sin(t)**, and again adding no canvas width or height control list.

```
(%i2) plot2d ([parametric, cos(t), sin(t), [t,-%pi,%pi]])$
```

This produces

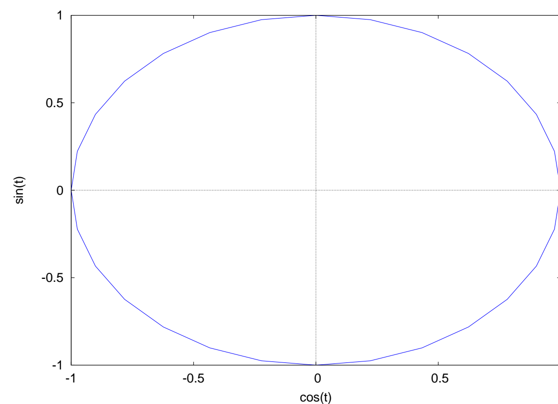

Figure 5: plot2d ([parametric, cos(t), sin(t), [t, -%pi, %pi]])

Now let's add a canvas width control list:

```
(%i3) plot2d ([parametric, cos(t), sin(t), [t, -%pi, %pi] ], [x,-4/3,4/3])$
```

which produces a "rounder" circle:

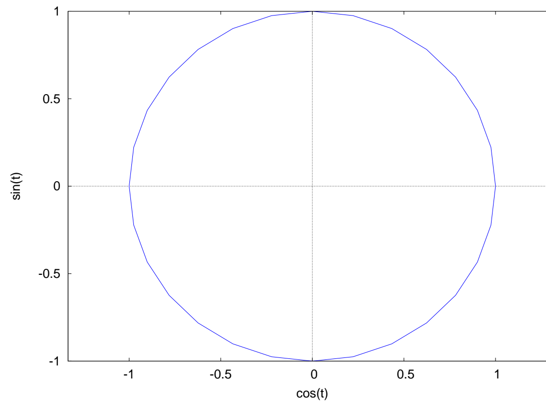

Figure 6: As above, but adding option [x, -4/3, 4/3]

The following versions produce precisely the same parametric plot:

```
(%i4) plot2d ([parametric, cos(x), sin(x), [x,-%pi,%pi]],
                [x,-4/3,4/3])$
(%i5) plot2d ([parametric, cos(y), sin(y), [y,-%pi,%pi]],
                [x,-4/3,4/3])$
```

An alternative method of producing a "round" circle with a parametric plot is to make use of the **gnuplot_preamble** option in the form:

```
(%i6) plot2d ([parametric, cos(t), sin(t), [t,-%pi,%pi]],
             [x,-1.2,1.2], [y,-1.2,1.2],
            [gnuplot_preamble,"set size ratio 1;"])$
```

which produces

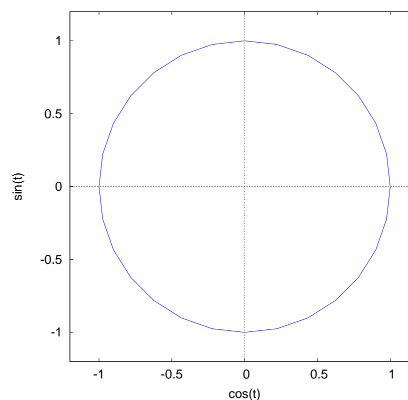

Figure 7: Using "set size ratio 1"

We now make a plot consisting of two plot objects, the first being the explicit expression **u^3**, and the second being the parametric plot object used above. We now need the syntax

```
plot2d ([plot-object-1, plot-object-2], possibly-required-draw-range-control,
                        other-option-lists )
```

Here is an example:

```
(%i7) plot2d (
              [ u^3,
                [parametric, cos(t), sin(t), [t,-%pi,%pi]]],
                [u,-0.8,0.8], [x,-4/3,4/3] )$
```

in which **[u,-0.8,0.8]** is required to determine the drawing range of **u^3**, and this produces

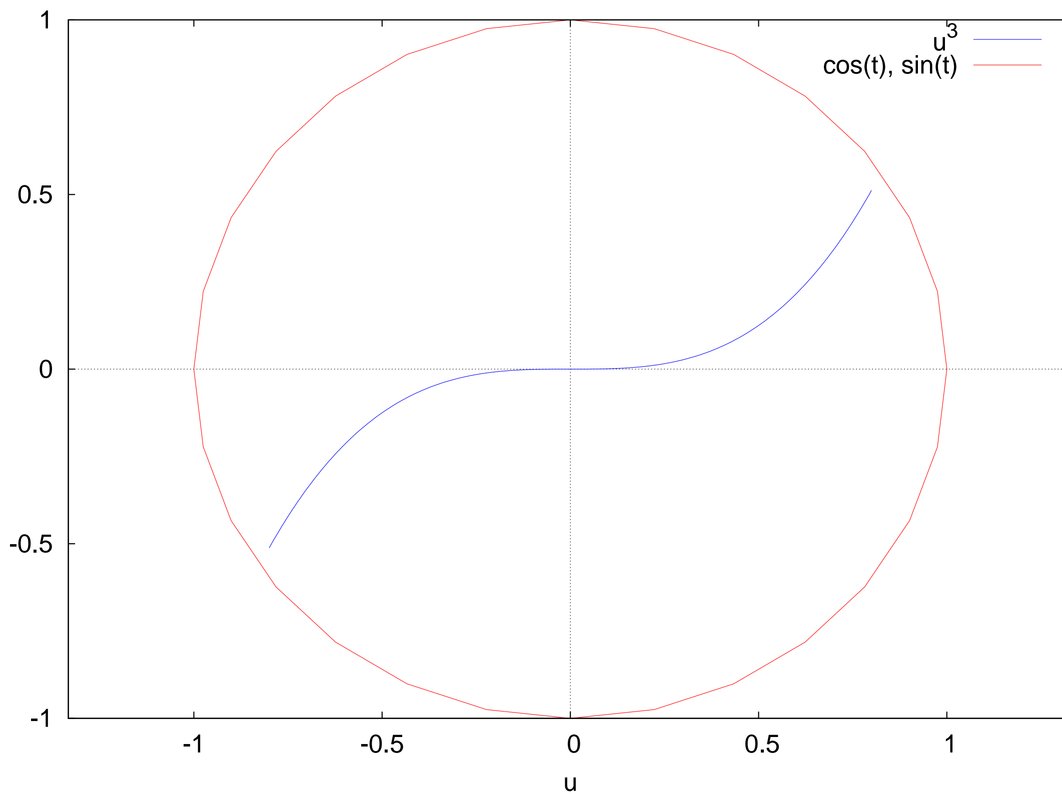

Figure 8: Combining an Explicit Expression with a Parametric Object

in which the horizontal axis label (by default) is **u**.

We now add a few more options to make this combined plot look a little better (more about some of these later).

```
(%i8) plot2d (
             [ [parametric, cos(t), sin(t), [t,-%pi,%pi]],u^3],
               [u,-1,1], [x,-1.2,1.2] , [y,-1.2,1.2], [nticks,200],
               [style, [lines,8]],  [xlabel," "], [ylabel," "],
               [box,false], [axes, false],
               [legend,false],[gnuplot_preamble,"set size ratio 1;"])$
```

which produces

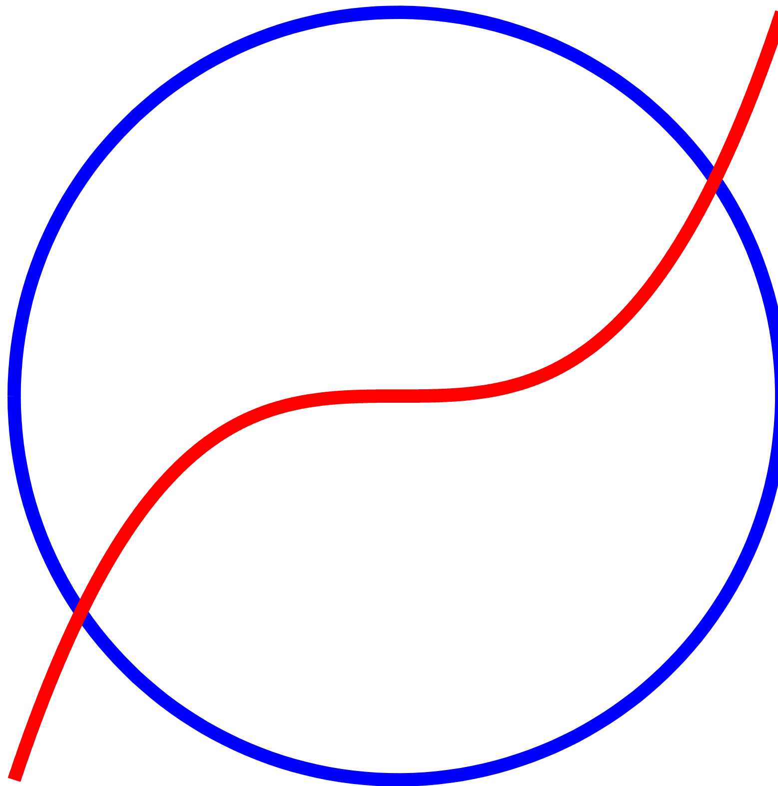

Figure 9: Drawing of $\mathbf{u}^3$ Over a Circle

The default value of **nticks** inside **plot2d** is `29`, and using `[nticks, 200]` yields a much smoother parametric curve.

2.1.3 Line Width and Color Controls

Each element to be included in the plot can have a separate **[lines, nlw, nlc]** entry in the **style** option list, with **nlw** determining the line width and **nlc** determining the line color. The default value of **nlw** is `1`, a very thin weak line. The use of **nlw = 5** creates a strong wider line.

The default values of **nlc** consist of a rotating color scheme which starts with **nlc = 1** (blue) and then progresses through nlc = 7 (black) and then repeats.

You will see the colors with the associated values of **nlc**, using the following code which draws a set of vertical lines in various colors. This code also shows an example of using **discrete** list objects, and the use of various options available.

```
(%i1) plot2d(
        [ [discrete,[[-4,-5],[-4,5]]], [discrete,[[-3,-5],[-3,5]]],
          [discrete,[[-2,-5],[-2,5]]],[discrete,[[-1,-5],[-1,5]]],
          [discrete,[[0,-5],[0,5]]],[discrete,[[1,-5],[1,5]]],
          [discrete,[[2,-5],[2,5]]],[discrete,[[3,-5],[3,5]]] ],

          [style,  [lines,10,0],[lines,10,1],[lines,10,2],
          [lines,10,3],[lines,10,4],[lines,10,5],[lines,10,6],
          [lines,10,7]],

          [x,-5,5],   [y,-7,7],
          [legend,"0","1","2","3","4","5","6","7"],
          [xlabel," "],   [ylabel," "],
          [box,false],[axes,false],
          [gnuplot_preamble, "set key bottom"] )$
```

Note that none of the objects being drawn are expressions or functions, so a draw parameter range list is not only not necessary but would make no sense, and that the optional width control list is `[x,-5,5]`.

The **plot2d** colors are: **0 = black, 1 = blue, 2 = red, 3 = violet**, **4 = dark green, 5 = dark brown, 6 = green, 7 = black, 8 = blue, ...**

We cannot use **plot2d** to produce an eps figure which will show the true **plot2d** colors since **plot2d** access to postscript **eps** file colors are: **0 = light blue, 1 = blue, 2 = red**, **3 = violet, 4 = black, 5 = yellow, 6 = green, 7 = light blue**.

Thus the **plot2d eps** color set replaces dark green and dark brown with yellow and light blue (but with different numbers too).

The following color bar plot, which was produced using **qdraw** (see Ch. 5) shows roughly the color bar chart you will see using the usual interactive **plot2d** mode.

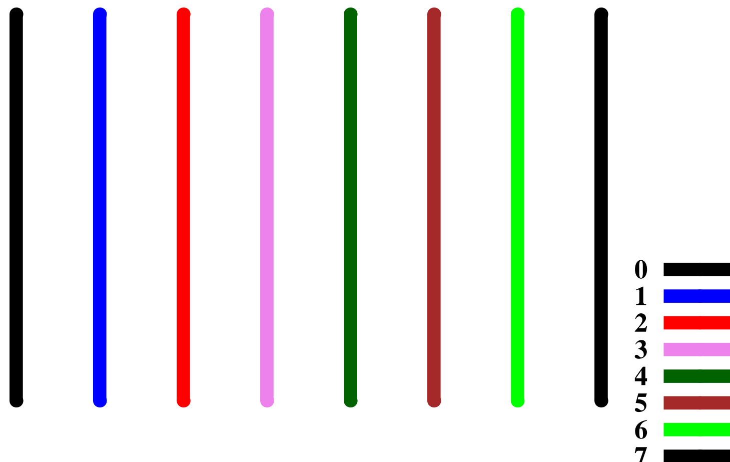

Figure 10: plot2d Color Numbers

For a simple example which uses color and line width controls, we plot the expressions **u^2** and **u^3** on the same canvas, using lines in black and red colors, and add a height control list, which has the syntax **[y, ymin, ymax]**.

```
(%i2) plot2d( [u^2,u^3],[u,0,2], [x, -.2, 2.5],
           [style, [lines,5,7],[lines,5,2]],
            [y,-1,4] )$
plot2d: expression evaluates to non-numeric value somewhere in plotting range.
```

A bug (v. 5.18.1) in **plot2d** incorrectly treats numbers outside the vertical range which has been set with **[y, -1, 4]** as if they were non-numeric objects.

The width and height control list parameters have been chosen to make it easy to see where the two curves cross for positive **u**. If you move the cursor over the crossing point, you can read off the coordinates from the cursor position printout in the lower left corner of the plot window.

This produces the plot:

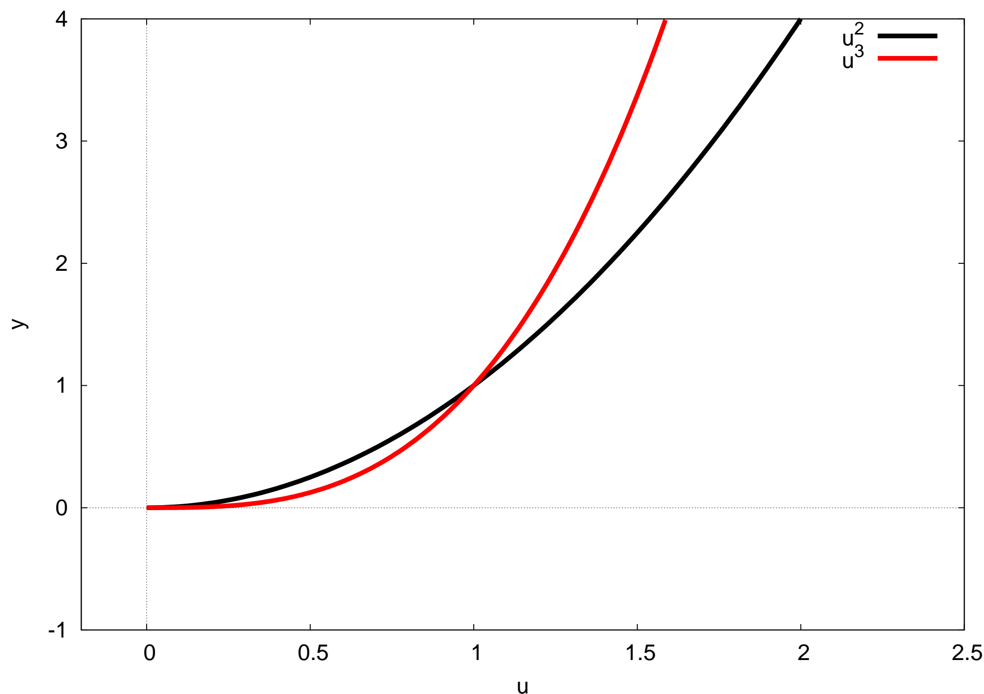

Figure 11: Black and Red Curves

This figure was included in this chapter by creating an eps file version (for inclusion in a latex file) using the code

```
(%i3)  plot2d( [u^2, u^3],[u,0,2], [x, -.2, 2.5],
           [style, [lines,10,4],[lines,10,2]],
            [y,-1,4],[psfile, "ch2p11.eps"] )$
```

This eps file version creates the key legend $\mathbf{u^2}$ instead of the **plot2d** console window version which has **u^2** as the legend.

Quoted Symbol Prevents Serious Error

Since **plot2d** has a special role for the symbols **x** and **y**, used in the optional width and height control lists, there is the danger that you might have assigned an expression or value to the symbol **y** (say) in your previous work, and since **plot2d** evaluates its arguments, you will get an error. Here is an example:

```
(%i1) y : x^2$
(%i2) plot2d ( u^3, [u,-1,1], [x,-2,2], [y,-10,10])$

[x^2,-10,10] is not a plot option.  Must be [symbol,..data]
 -- an error.  To debug this try debugmode(true);
(%i3) plot2d ( u^3, [u,-1,1], [x,-2,2], ['y,-10,10])$
(%i4) 'y;
(%o4)                              y
(%i5) y;
                                    2
(%o5)                              x
(%i6) ev(y);
                                    2
(%o6)                              x
```

By using the single quoted symbol **'y** in the canvas height control list, you are preventing **plot2d** from "evaluating" the symbol and assigning the first slot of that list to the expression **x^2**. You can use the single quote **'** for the first slot of both control lists:

```
(%i7) plot2d ( u^3, [u,-1,1], ['x,-2,2], ['y,-10,10])$
```

and the plot appears with no problems.

2.1.4 Discrete Data Plots: Point Size, Color, and Type Control

We have seen some simple parametric plot examples above. Here we make a more elaborate plot which includes discrete data points which locate on the curve special places, with information on the key legend about those special points. We force large size points with special color choices, using the maximum amount of control in the
[points, nsize, ncolor, ntype] style assignments.

```
(%i1) obj_list : [ [parametric, 2*cos(t),t^2,[t,0,2*%pi]],
                     [discrete,[[2,0]]],[discrete,[[0,(%pi/2)^2]]],
              [discrete,[[-2,%pi^2]]],[discrete,[[0,(3*%pi/2)^2]]] ]$
(%i2) style_list : [style, [lines,4,7],[points,5,1,1],[points,5,2,1],
                 [points,5,6,1],[points,5,3,1]]$
(%i3) legend_list : [legend, " ","t = 0","t = pi/2","t = pi",
                " t = 3*pi/2"]$
(%i4) plot2d( obj_list, [x,-3,4], [y,-1,40],style_list,
                  [xlabel,"X = 2 cos( t ), Y = t ^2 "],
                   [ylabel, " "] ,legend_list )$
```

This produces the plot:

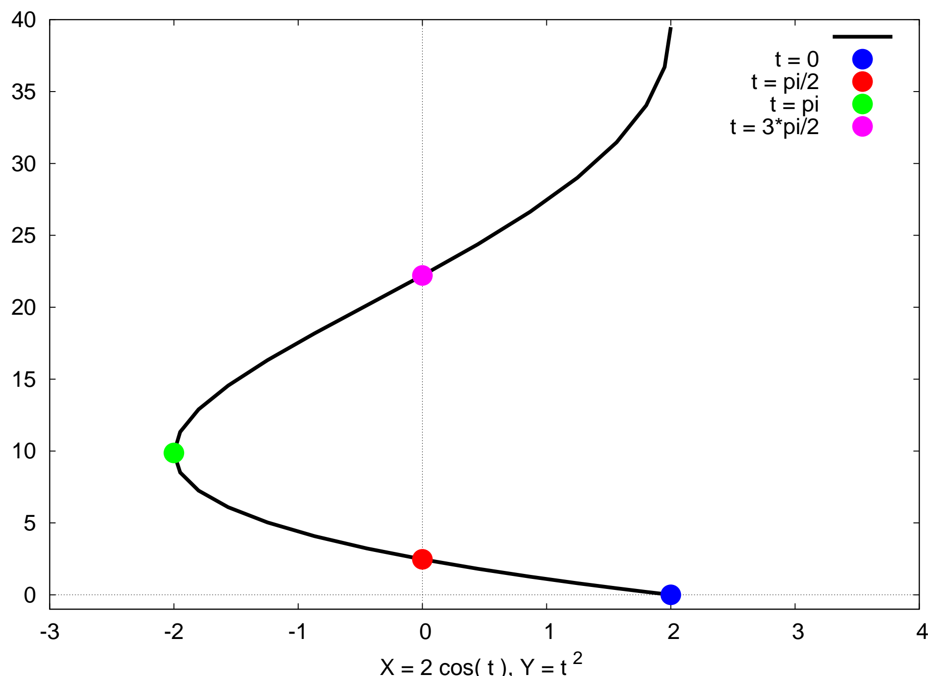

Figure 12: Parametric Plot with Discrete Points

The **points** style option has any of the following forms: `[points]`, or `[points, point_size]`, or `[points, point_size, point_color]`, or `[points, point_size, point_color, point_type]`, in order of increasing control. The default point size is given by the integer **1** which is small. The default point colors are the same cyclic colors used by the **lines** style. The default point type is a cyclic order starting with **2 = open circle** , then continuing **3 = plus sign**, **4 = diagonal crossed lines as capital X**, **5 = star**, **6 = filled square**, 7 = open square, 8 = filled triangle point up, **9 = open triangle point up**, **10 = filled triangle point down**, **11 = open triangle point down 12 = filled diamond**, **13 = open diamond = 0**, **14 = filled circle**, which is the same as **1**. Thus if you use `[points,5]` you will get good size points, and both the color and shape will cycle through the default order. If you use `[points,5,2]` you will force a red color but the shape will depend on the contents and order of the rest of the objects list.

Next we combine a list of twelve (x,y) pairs of points with the key word **discrete** to form a discrete object type for **plot2d**, and then look at the data points without adding the optional canvas width control.

```
(%i5) data_list : [discrete,
         [ [1.1,-0.9],[1.45,-1],[1.56,0.3],[1.88,2],
              [1.98,3.67],[2.32,2.6],[2.58,1.14],
           [2.74,-1.5],[3,-0.8],[3.3,1.1],
           [3.65,0.8],[3.72,-2.9] ] ]$
(%i6) plot2d( data_list, [style,[points]])$
```

This produces the plot

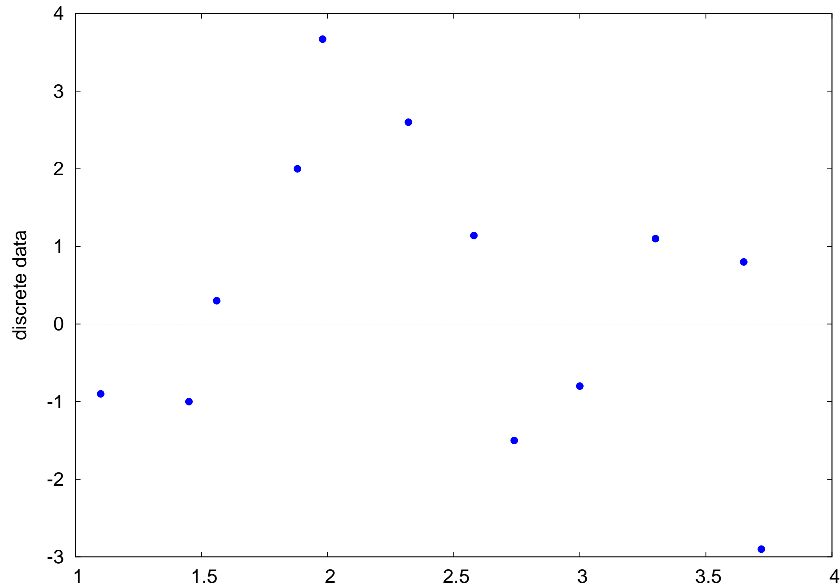

Figure 13: Twelve Data Points

We now combine the data points with a curve which is a possible fit to these data points over the draw parameter range **[u, 1, 4]**.

```
(%i7) plot2d( [sin(u)*cos(3*u)*u^2, data_list],
               [u,1,4],  [x,0,5],[y,-10,8],
          [style,[lines,4,1],[points,4,2,1]])$
plot2d: expression evaluates to non-numeric value somewhere in plotting range.
```

which produces the plot

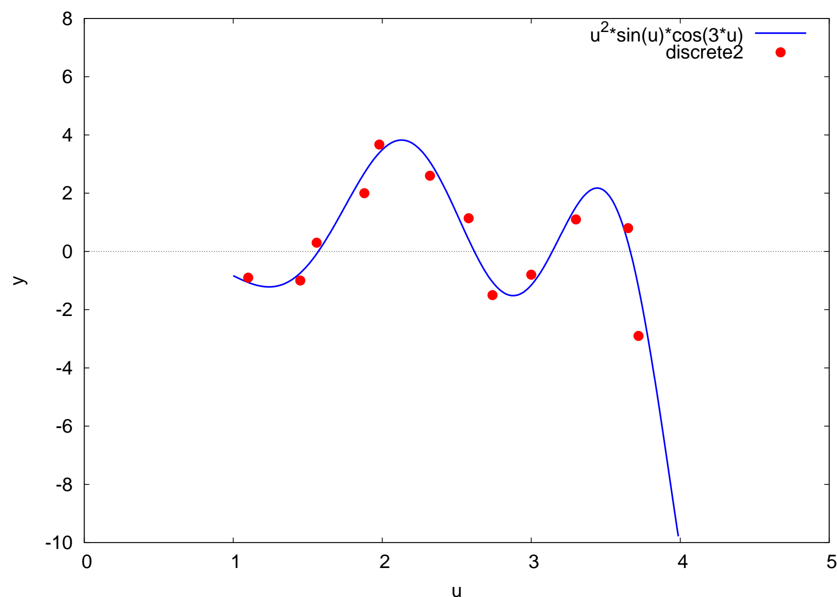

Figure 14: Curve Plus Data

2.1.5 More gnuplot_preamble Options

Here is an example of using the **gnuplot_preamble** options to add a grid, a title, and position the plot key at the bottom center of the canvas. Note the use of a semi-colon between successive gnuplot instructions.

```
(%i1) plot2d([ u*sin(u),cos(u)],[u,-4,4] ,[x,-8,8],
              [style,[lines,5]],
         [gnuplot_preamble,"set grid; set key bottom center;
               set title 'Two Functions';"])$
```

which produces the plot

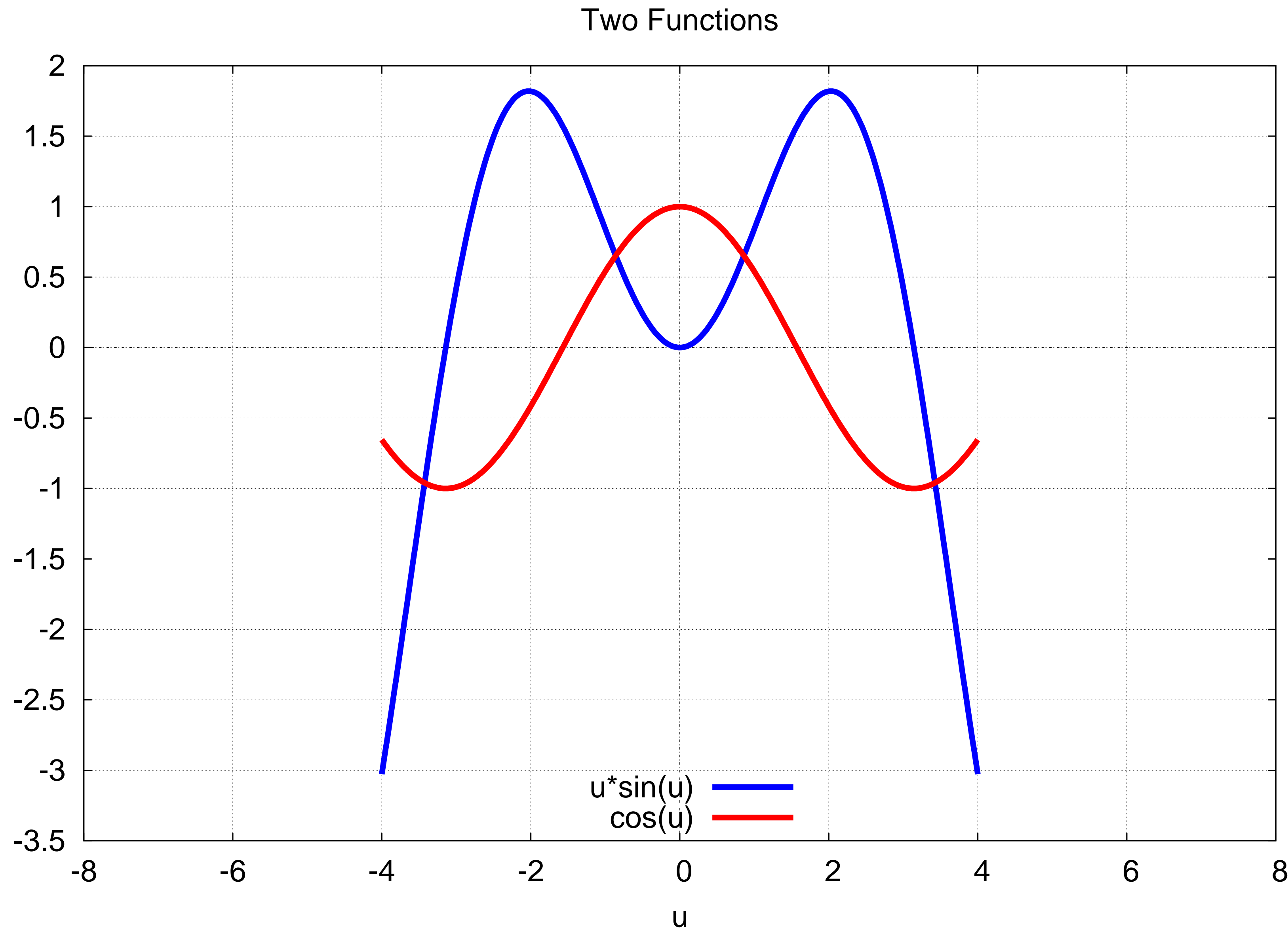

Figure 15: Using the gnuplot_preamble Option

Another option we have previously used is **`set zeroaxis lw 2;`** to get more prominent **`x`** and **`y`** axes. Another example of key location would be **`set key top left;`**. We have also previously used **`set size ratio 1;`** to get a "rounder" circle.

2.1.6 Using qplot for Quick Plots of One or More Functions

The file **mbe1util.mac** is posted with Ch. 1. This "utility file" contains a function called **qplot** which can be used for quick plotting of functions in place of **plot2d**.

The function **qplot** (**q** for "quick") accepts the default cyclic colors but always uses thicker lines than the **plot2d** default, adds more prominent x and y axes to the plot, and adds a grid. Here are some examples of use. (We include use with **discrete** lists only for completeness, since there is no way to get the **points** style with **qplot**.)

```
(%i1) load(mbe1util);
(%o1)                        c:/work2/mbe1util.mac
(%i2) qplot(sin(u),[u,-%pi,%pi])$
(%i3) qplot(sin(u),[u,-%pi,%pi],[x,-4,4])$
(%i4) qplot(sin(u),[u,-%pi,%pi],[x,-4,4],[y,-1.2,1.2])$
(%i5) qplot([sin(u),cos(u)],[u,-%pi,%pi])$
(%i6) qplot([sin(u),cos(u)],[u,-%pi,%pi],[x,-4,4])$
(%i7) qplot([sin(u),cos(u)],[u,-%pi,%pi],[x,-4,4],[y,-1.2,1.2])$
(%i8) qplot ([parametric, cos(t), sin(t), [t,-%pi,%pi]],
             [x,-2.1,2.1],[y,-1.5,1.5])$
```

The last use involved only a parametric object, and the "prange" list is interpreted as a width control list based on the symbol **x**. The next example includes both an expression depending on the parameter **u** and a parametric object, so we must have a draw parameter control list.

```
(%i9) qplot ([ u^3,
              [parametric, cos(t), sin(t), [t,-%pi,%pi]]],
              [u,-1,1],[x,-2.1,2.1],[y,-1.5,1.5])$
```

We have used **qdraw** (Ch.5) to produce an eps figure we can use here which roughly represents what you get from that last example:

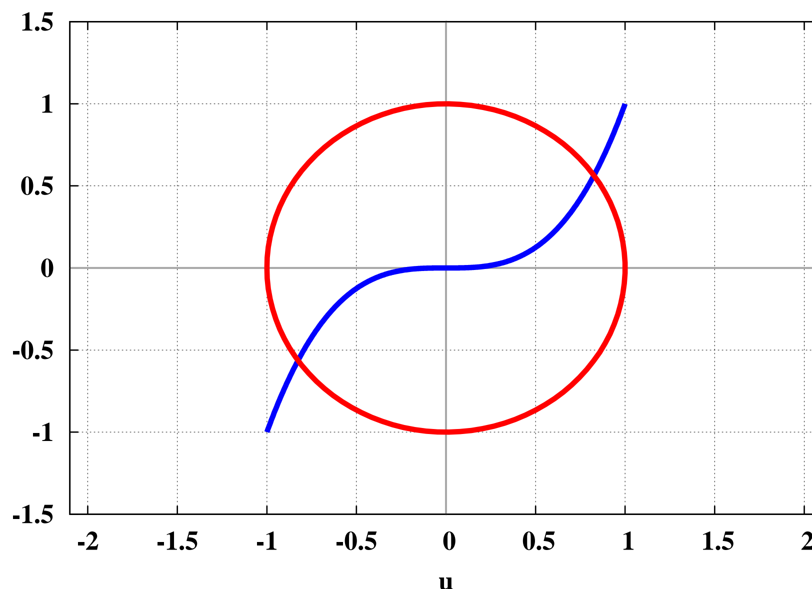

Figure 16: qplot example

Here are two **discrete** examples which draw vertical lines.

```
(%i10) qplot([discrete,[[0,-2],[0,2]]],[x,-2,2],[y,-4,4])$
(%i11) qplot( [ [discrete,[[-1,-2],[-1,2]]],
                 [discrete,[[1,-2],[1,2]]]],[x,-2,2],[y,-4,4])$
```

Here is the code (in mbe1util.mac) which defines the Maxima function **qplot**.

```
qplot ( exprlist, prange, [hvrange]) :=
    block([optlist, plist],
      optlist : [ [nticks,100], [legend, false],
           [ylabel, " "], [gnuplot_preamble, "set grid; set zeroaxis lw 2;"] ],
      optlist : cons ( [style,[lines,5]], optlist ),
      if length (hvrange) = 0 then plist : []
            else   plist : hvrange,
      plist : cons (prange,plist),
      plist : cons (exprlist,plist),
      plist : append ( plist, optlist ),
      apply (plot2d, plist ) )$
```

In this code, the third argument is an optional argument. The local **plist** accumulates the arguments to be passed to **plot2d** by use of **cons** and **append**, and is then passed to **plot2d** by the use of **apply**. The order of using **cons** makes sure that **exprlist** will be the first element, (and **prange** will be the second) seen by **plot2d**. In this example you can see several tools used for programming with lists.

Several choices have been made in the **qplot** code to get quick and uncluttered plots of one or more functions. One choice was to add a grid and stronger `x` and `y` axis lines. Another choice was to eliminate the key legend by using the option `[legend, false]`. If you want a key legend to appear when plotting multiple functions, you should remove that option from the code and reload **mbe1util.mac**.

2.2 Least Squares Fit to Experimental Data

2.2.1 Maxima and Least Squares Fits: lsquares_estimates

Suppose we are given a list of **[x,y]** pairs which are thought to be roughly described by the relation **y = a*x^b + c**, where the three parameters are all of order **1**. We can use the data of **[x,y]** pairs to find the "best" values of the unknown parameters **[a, b, c]**, such that the data is described by the equation **y = a*x^b + c** (a three parameter fit to the data).

We are using one of the Manual examples for **lsquares_estimates**.

```
(%i1) dataL : [[1, 1], [2, 7/4], [3, 11/4], [4, 13/4]];

                                      7       11       13
(%o1)                  [[1, 1], [2, -], [3, --], [4, --]]
                                      4       4        4
(%i2) dataM : apply ('matrix, dataL);
                                     [ 1  1  ]
                                     [       ]
                                     [    7  ]
                                     [ 2  -  ]
                                     [    4  ]
                                     [       ]
(%o2)                                [    11 ]
                                     [ 3  -- ]
                                     [    4  ]
                                     [       ]
                                     [    13 ]
                                     [ 4  -- ]
                                     [    4  ]
(%i3) load (lsquares)$
(%i4) fpprintprec:8$
(%i5) lsquares_estimates (dataM, [x,y], y=a*x^b+c,
                [a,b,c], initial=[3,3,3], iprint=[-1,0] );
(%o5)             [[a = 1.3873659, b = 0.711096, c = - 0.414271]]
(%i6) fit : a*x^b + c , % ;
                                     0.711096
(%o6)                     1.3873659 x         - 0.414271
```

Note that we must use **load (lsquares);** to use this method. We can now make a plot of both the discrete data points and the least squares fit to those four data points.

```
(%i7) plot2d ([fit,[discrete,dataL]],[x,0,5],
        [style,[lines,5],[points,4,2,1]],
         [legend,"fit", "data"],
         [gnuplot_preamble,"set key bottom;"])$
```

which produces the plot

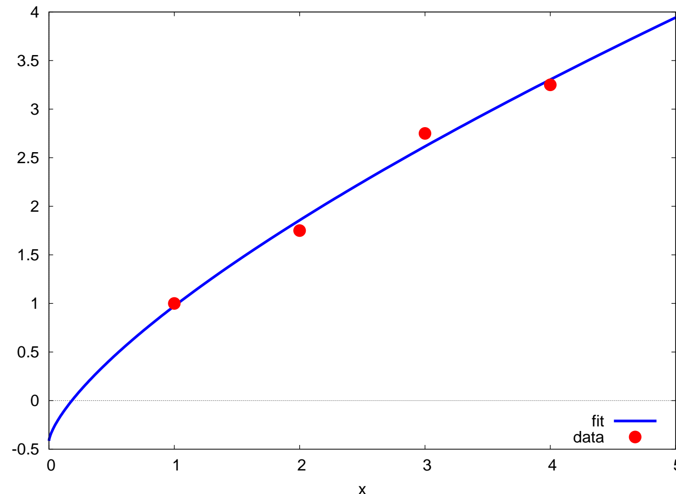

Figure 17: Three Parameter Fit to Four Data Points

2.2.2 Syntax of lsquares_estimates

The minimal syntax is

```
lsquares_estimates (data-matrix, data-variable-list, fit-eqn, param-list );
```

in which the **data-variable-list** assigns a variable name to the corresponding column of the **data-matrix**, and the **fit-eqn** is an equation which is a relation among the data variable symbols and the equation parameters which appear in **param-list**. The function returns the "best fit" values of the equation parameters in the form **[[p1 = p1val, p2 = p2val, ...]]**.

In the example above, the data variable list was **[x, y]** and the parameter list was **[a, b, c]**.

If an exact solution cannot be found, a numerical approximation is attempted using **lbfgs**, in which case, all the elements of the data matrix should be "numbers" **numberp(x) -> true**. This means that **%pi** and **%e**, for example, should be converted to explicit numbers before use of this method.

```
(%i1) expr : 2*%pi + 3*exp(-4);
                                                 - 4
(%o1)                         2 %pi + 3 %e
(%i2) listconstvars:true$
(%i3) listofvars(expr);
(%o3)                            [%e, %pi]
(%i4) map('numberp,%);
(%o4)                         [false, false]
(%i5) fullmap('numberp,expr);
                              true
(%o5)                   false      true + false true
(%i6) float(expr);
(%o6)                       6.338132223845789
(%i7) numberp(%);
(%o7)                                true
```

Optional arguments to **lsquares_estimates** are (in any order)

```
initial = [p10, p20,...], iprint = [n, m], tol = search-tolerance
```

The list `[p10, p20, ...]` is the optional list of initial values of the equation parameters, and without including your own guess for starting values this list defaults (in the code) to `[1, 1, ...]`.

The first integer `n` in the `iprint` list controls how often progress messages are printed to the screen. The default is `n = 1` which causes a new progress message printout each iteration of the search method. Using `n = -1` surpresses all progress messages. Using `n = 5` allows one progress message every five iterations.

The second integer `m` in the `iprint` list controls the verbosity, with `m = 0` giving minimum information and `m = 3` giving maximum information.

The option `iprint = [-1,0]` will hide the details of the search process.

The default value of the `search-tolerance` is `1e-3`, so by using the option `tol = 1e-8` you might find a more accurate solution.

Many examples of the use of the **lsquares** package are found in the file `lsquares.mac`, which is found in the `...share/contrib` folder. You can also see great examples of efficient programming in the Maxima language in that file.

2.2.3 Coffee Cooling Model

"Newton's law of cooling" (only approximate and not a law) assumes the rate of decrease of temperature (celsius degrees per minute) is proportional to the instantaneous difference between the temperature $\mathbf{T(t)}$ of the coffee in the cup and the surrounding ambient temperature $\mathbf{T_s}$, the latter being treated as a constant. If we use the symbol $\mathbf{r}$ for the "rate constant" of proportionality, we then assume the cooling of the coffee obeys the first order differential equation

$$\frac{\mathbf{d\,T}}{\mathbf{d\,t}} = -\mathbf{r}\,(\mathbf{T(t)} - \mathbf{T_s}) \tag{2.1}$$

Since $\mathbf{T}$ has dimension degrees Celsius, and $\mathbf{t}$ has dimension minute, the dimension of the rate constant $\mathbf{r}$ must be `1/min`.

(This attempt to employ a rough mathematical model which can be used for the cooling of a cup of coffee avoids a bottom-up approach to the problem, which would require mathematical descriptions of the four distinct physical mechanisms which contribute to the decrease of thermal energy in the system hot coffee plus cup to the surroundings: thermal radiation (net electromagnetic radiation flux, approximately black body) energy transport across the surface of the liquid and cup, collisional heat conduction due to the presence of the surrounding air molecules, convective energy transport due to local air temperature rise, and finally evaporation which is the escape of the fastest coffee molecules which are able to escape the binding surface forces at the liquid surface. If the temperature difference between the coffee and the ambient surroundings is not too large, experiment shows that the simple relation above is roughly true.)

This differential equation is easy to solve "by hand" , since we can write

$$\frac{\mathbf{d\,T}}{\mathbf{d\,t}} = \frac{\mathbf{d\,(T - T_s)}}{\mathbf{d\,t}} = \frac{\mathbf{d\,y}}{\mathbf{d\,t}} \tag{2.2}$$

and then divide both sides by $\mathbf{y = (T - T_s)}$, multiply both sides by $\mathbf{d\,t}$, and use $\mathbf{d\,y/y = d\ln(y)}$ and finally integrate both sides over corresponding intervals to get $\mathbf{\ln(\mathit{y}) - \ln(\mathit{y_0}) = \ln(y/y_0) = -r\,t}$, where $\mathbf{y_0 = T(0) - T_s}$ involves the initial temperature at $\mathbf{t = 0}$. Since

$$\mathbf{e^{\ln(A)} = A}, \tag{2.3}$$

by equating the exponential of the left side to that of the right side, we get

$$\mathbf{T(t) = T_s + (T(0) - T_s)\, e^{-r\,t}}. \tag{2.4}$$

Using **ode2**, **ic1**, **expand**, and **collectterms**, we can also use Maxima just for fun:

```
(%i1) de : 'diff(T,t) + r*(T - Ts);
                                 dT
(%o1)                            -- + r (T - Ts)
                                 dt
(%i2) gsoln : ode2(de,T,t);
                                 - r t    r t
(%o2)                      T = %e      (%e    Ts + %c)
(%i3) de, gsoln, diff, ratsimp;
(%o3)                                   0
(%i4) ic1 (gsoln, t = 0, T = T0);
                              - r t          r t
(%o4)                   T = %e      (T0 + (%e    - 1) Ts)
(%i5) expand (%);
                              - r t        - r t
(%o5)                   T = %e      T0 - %e      Ts + Ts
(%i6) Tcup : collectterms ( rhs(%), exp(-r*t) );
                               - r t
(%o6)                        %e      (T0 - Ts) + Ts
(%i7) Tcup, t = 0;
(%o7)                                  T0
```

We arrive at the same solution as found ”by hand”. We have checked the particular solution for the initial condition and checked that our original differential equation is satisfied by the general solution.

2.2.4 Experiment Data: file_search, printfile, read_nested_list, and makelist

Let’s take some ”real world” data for this problem (p. 21, An Introduction to Computer Simulation Methods, 2nd ed., Harvey Gould and Jan Tobochnik, Addison-Wesley, 1996) which is in a data file `c:\work2\coffee.dat` on the author’s Window’s XP computer (data file available with this chapter on the author’s webpage).

This file contains three columns of tab separated numbers, column one being the elapsed time in minutes, column two the Celsius temperature of the system glass plus coffee for black coffee, and column three the Celsius temperature for the case glass plus creamed coffee. The glass-coffee temperature was recorded with an estimated accuracy of $\mathbf{0.1^{\circ}C}$. The ambient temperature of the surroundings was $\mathbf{17\,^{\circ}C}$.

We need to use double quotes for a file name which includes an extension like `.dat`.

```
(%i1) file_search("coffee.dat");
(%o1)                              coffee.dat
```

No path was needed because of the contents of `maxima-init.mac`. We can look at the whole file at one burst with **printfile**. (We will show only the top of the output.)

```
(%i2) printfile("coffee.dat");
0        82.3        68.8
2        78.5        64.8
4        74.3        62.1
6        70.7        59.9
8        67.6        57.7
----------------
  etc, etc.
  ---------
```

We now use **read_nested_list** which will create a list of sublists, one sublist per row.

```
(%i3) fpprintprec:6$
(%i4) cdata : read_nested_list("coffee.dat");
(%o4) [[0, 82.3, 68.8], [2, 78.5, 64.8], [4, 74.3, 62.1], [6, 70.7, 59.9],
[8, 67.6, 57.7], [10, 65.0, 55.9], [12, 62.5, 53.9], [14, 60.1, 52.3],
[16, 58.1, 50.8], [18, 56.1, 49.5], [20, 54.3, 48.1], [22, 52.8, 46.8],
[24, 51.2, 45.9], [26, 49.9, 44.8], [28, 48.6, 43.7], [30, 47.2, 42.6],
[32, 46.1, 41.7], [34, 45.0, 40.8], [36, 43.9, 39.9], [38, 43.0, 39.3],
[40, 41.9, 38.6], [42, 41.0, 37.7], [44, 40.1, 37.0], [46, 39.5, 36.4]]
```

We now use **makelist** to create a (time, temperature) list based on the black coffee data and then based on the white (creamed coffee) data.

```
(%i5) black_data : makelist( [first(cdata[i]),second(cdata[i])],
                        i,1,length(cdata));
(%o5) [[0, 82.3], [2, 78.5], [4, 74.3], [6, 70.7], [8, 67.6], [10, 65.0],
[12, 62.5], [14, 60.1], [16, 58.1], [18, 56.1], [20, 54.3], [22, 52.8],
[24, 51.2], [26, 49.9], [28, 48.6], [30, 47.2], [32, 46.1], [34, 45.0],
[36, 43.9], [38, 43.0], [40, 41.9], [42, 41.0], [44, 40.1], [46, 39.5]]
(%i6) white_data : makelist( [first(cdata[i]),third(cdata[i])],
                        i,1,length(cdata));
(%o6) [[0, 68.8], [2, 64.8], [4, 62.1], [6, 59.9], [8, 57.7], [10, 55.9],
[12, 53.9], [14, 52.3], [16, 50.8], [18, 49.5], [20, 48.1], [22, 46.8],
[24, 45.9], [26, 44.8], [28, 43.7], [30, 42.6], [32, 41.7], [34, 40.8],
[36, 39.9], [38, 39.3], [40, 38.6], [42, 37.7], [44, 37.0], [46, 36.4]]
```

2.2.5 Least Squares Fit of Coffee Cooling Data

We now use **lsquares_estimates** to use a least squares fit with each of our data sets to our phenomenological model, that is finding the "best" value of the cooling rate constant **r** that appears in Eq. (2.4). The function **lsquares_estimates(data_matrix, eqnvarlist,eqn,paramlist)** is available after using **load(lsquares)**.

To save space in this chapter we use **display2d:false** to surpress the default two dimensional display of a Maxima **matrix** object.

```
(%i7) display2d:false$
(%i8) black_matrix : apply ( 'matrix, black_data);
(%o8) matrix([0,82.3],[2,78.5],[4,74.3],[6,70.7],[8,67.6],[10,65.0],[12,62.5],
             [14,60.1],[16,58.1],[18,56.1],[20,54.3],[22,52.8],[24,51.2],
             [26,49.9],[28,48.6],[30,47.2],[32,46.1],[34,45.0],[36,43.9],
             [38,43.0],[40,41.9],[42,41.0],[44,40.1],[46,39.5])
(%i9) white_matrix : apply ( 'matrix, white_data);
(%o9) matrix([0,68.8],[2,64.8],[4,62.1],[6,59.9],[8,57.7],[10,55.9],
              [12,53.9],[14,52.3],[16,50.8],[18,49.5],[20,48.1],[22,46.8],
              [24,45.9],[26,44.8],[28,43.7],[30,42.6],[32,41.7],[34,40.8],
              [36,39.9],[38,39.3],[40,38.6],[42,37.7],[44,37.0],[46,36.4])
```

We now load **lsquares.mac** and calculate the "best fit" values of the cooling rate constant **r** for both cases. For the black coffee case, `T0 = 82.3 deg C` and `Ts = 17 deg C` and we surpress the units.

```
(%i10) display2d:true$
(%i11) load(lsquares);
(%o11) C:/PROGRA~1/MAXIMA~3.1/share/maxima/5.18.1/share/contrib/lsquares.mac
(%i12) black_eqn : T = 17 + 65.3*exp(-r*t);
                                          - r t
(%o12)                       T = 65.3 %e      + 17
```

```
(%i13) lsquares_estimates ( black_matrix, [t,T], black_eqn, [r],
                              iprint = [-1,0] );
(%o13)                         [[r = 0.02592]]
(%i14) black_fit : rhs ( black_eqn ), %;
                                    - 0.02592 t
(%o14)                       65.3 %e            + 17
```

Thus **rblack** is roughly `0.026 min^(-1)`.

For the white coffee case, `T0 = 68.8 deg C` and `Ts = 17 deg C`.

```
(%i15) white_eqn : T = 17 + 51.8*exp(-r*t);
                                         - r t
(%o15)                         T = 51.8 %e      + 17
(%i16) lsquares_estimates ( white_matrix, [t,T], white_eqn, [r],
                             iprint = [-1,0] );
(%o16)                         [[r = 0.0237]]
(%i17) white_fit : rhs ( white_eqn ), %;
                                    - 0.0237 t
(%o17)                       51.8 %e           + 17
```

Thus **rwhite** is roughly `0.024 min^(-1)`, a slightly smaller value than for the black coffee (which is reasonable since a black body is a better radiator of thermal energy than a white surface).

A prudent check on mathematical reasonableness can be made by using, say, the two data points for $\mathbf{t} = \mathbf{0}$ and $\mathbf{t} = \mathbf{24}\,\mathbf{min}$ to solve for a rough value of **r**. For this rough check, the author concludes that **rblack** is roughly `0.027 min^(-1)` and **rwhite** is roughly `0.024 min^(-1)`.

We can now plot the temperature data against the best fit model curve, first for the black coffee case.

```
(%i18) plot2d( [ black_fit ,[discrete,black_data] ],
           [t,0,50], [style, [lines,5], [points,2,2,6]],
           [ylabel," "] ,
           [xlabel," Black Coffee T(deg C) vs. t(min) with r = 0.026/min"],
           [legend,"black fit","black data"] )$
```

which produces the plot

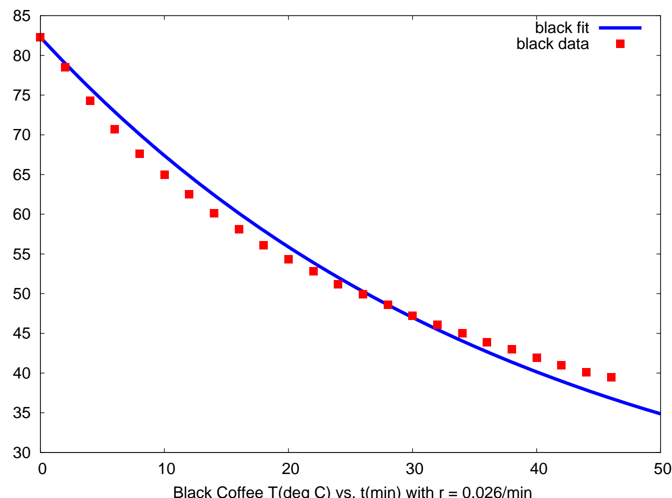

Figure 18: Black Coffee Data and Fit

and next plot the white coffee data and fit:

```
(%i19) plot2d( [ white_fit ,[discrete, white_data] ],
          [t,0,50], [style, [lines,5], [points,2,2,6]],
          [ylabel," "] ,
          [xlabel," White Coffee T(deg C) vs. t(min) with r = 0.024/min"],
          [legend,"white fit","white data"] )$
```

which yields the plot

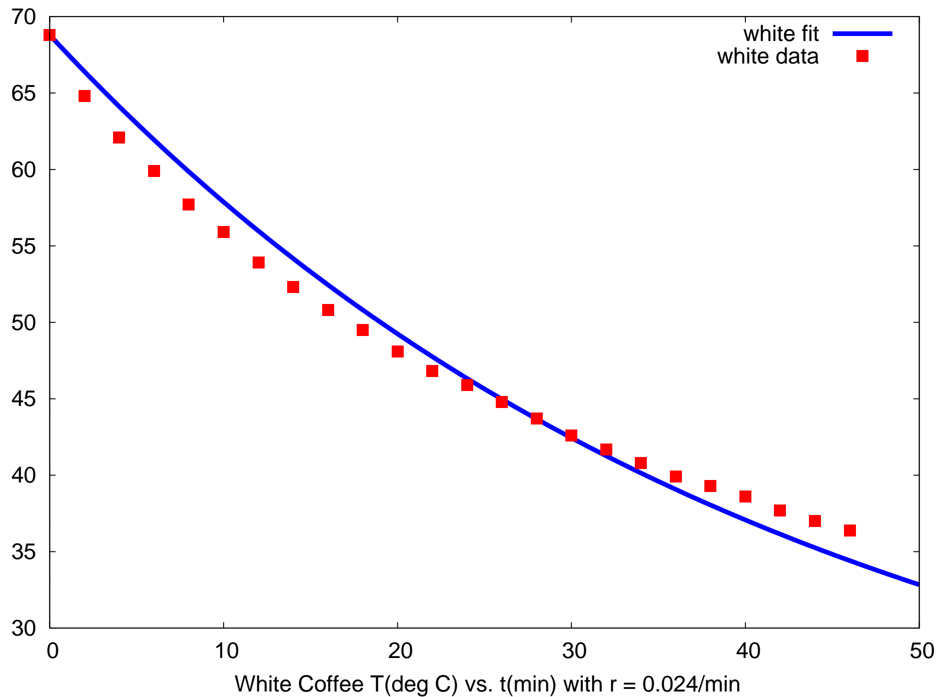

Figure 19: White Coffee Data and Fit

Cream at Start or Later?

Let's use the above approximate values for the cooling rate constants to find the fastest method to use to get the temperature of hot coffee down to a drinkable temperature. Let's assume we start with a glass of very hot coffee, $\mathbf{T_0 = 90^\circ C}$, and want to compare two methods of getting the temperature down to $\mathbf{75^\circ C}$, which we assume is low enough to start sipping. We will assume that adding cream lowers the temperature of the coffee by $\mathbf{5^\circ C}$ for both options we explore. Option 1 (white option) is to immediately add cream and let the creamed coffee cool down from $\mathbf{85^\circ C}$ to $\mathbf{75^\circ C}$. We first write down a general expression as a function of **T0** and **r**, and then substitute values appropriate to the white coffee cooldown.

```
(%i20) T : 17 + (T0 -17)*exp(-r*t);
                              - r t
(%o20)                      %e      (T0 - 17) + 17
(%i21) T1 : T, [T0 = 85, r = 0.0237];
                                 - 0.0237 t
(%o21)                       68 %e           + 17
(%i22) t1 : find_root(T1 - 75,t,2,10);
(%o22)                                 6.71159
```

The "white option" requires about $\mathbf{6.7\,min}$ for the coffee to be sippable.

Option 2 (the black option) lets the black coffee cool from $90°\mathrm{C}$ to $80°\mathrm{C}$, and then adds cream, immediately getting the temperature down from $80°\mathrm{C}$ to $75°\mathrm{C}$

```
(%i23) T2 : T, [T0 = 90, r = 0.02592];
                                    - 0.02592 t
(%o23)                         73 %e            + 17
(%i24) t2 : find_root(T2 - 80,t,2,10);
(%o24)                                5.68382
```

The black option (option 2) is the fastest method to cool the coffee, taking about $5.68\,\mathrm{min}$ which is about $62\,\mathrm{sec}$ less than the white option 1.

Maxima by Example:
Ch. 3, Ordinary Differential Equation Tools *

Edwin L. Woollett

September 16, 2010

Contents

3.1 Solving Ordinary Differential Equations . . . 3
3.2 Solution of One First Order Ordinary Differential Equation (ODE) . . . 3
- 3.2.1 Summary Table . . . 3
- 3.2.2 Exact Solution with **ode2** and **ic1** . . . 3
- 3.2.3 Exact Solution Using **desolve** . . . 5
- 3.2.4 Numerical Solution and Plot with **plotdf** . . . 6
- 3.2.5 Numerical Solution with 4th Order Runge-Kutta: **rk** . . . 7

3.3 Solution of One Second Order ODE or Two First Order ODE's . . . 9
- 3.3.1 Summary Table . . . 9
- 3.3.2 Exact Solution with **ode2**, **ic2**, and **eliminate** . . . 9
- 3.3.3 Exact Solution with **desolve, atvalue,** and **eliminate** . . . 12
- 3.3.4 Numerical Solution and Plot with **plotdf** . . . 16
- 3.3.5 Numerical Solution with 4th Order Runge-Kutta: **rk** . . . 17

3.4 Examples of ODE Solutions . . . 19
- 3.4.1 Ex. 1: Fall in Gravity with Air Friction: Terminal Velocity . . . 19
- 3.4.2 Ex. 2: One Nonlinear First Order ODE . . . 22
- 3.4.3 Ex. 3: One First Order ODE Which is Not Linear in Y' . . . 23
- 3.4.4 Ex. 4: Linear Oscillator with Damping . . . 24
- 3.4.5 Ex. 5: Underdamped Linear Oscillator with Sinusoidal Driving Force . . . 28
- 3.4.6 Ex. 6: Regular and Chaotic Motion of a Driven Damped Planar Pendulum . . . 30
- 3.4.7 Free Oscillation Case . . . 31
- 3.4.8 Damped Oscillation Case . . . 32
- 3.4.9 Including a Sinusoidal Driving Torque . . . 33
- 3.4.10 Regular Motion Parameters Case . . . 33
- 3.4.11 Chaotic Motion Parameters Case. . . . 37

3.5 Using **contrib_ode** for ODE's . . . 43

*This version uses **Maxima 5.18.1** Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions to `woollett@charter.net`

Preface

```
COPYING AND DISTRIBUTION POLICY
This document is part of a series of notes titled
"Maxima by Example" and is made available
via the author's webpage http://www.csulb.edu/~woollett/
to aid new users of the Maxima computer algebra system.

NON-PROFIT PRINTING AND DISTRIBUTION IS PERMITTED.

You may make copies of this document and distribute them
to others as long as you charge no more than the costs of printing.

These notes (with some modifications) will be published in book form
eventually via Lulu.com in an arrangement which will continue
to allow unlimited free download of the pdf files as well as the option
of ordering a low cost paperbound version of these notes.
```

Feedback from readers is the best way for this series of notes to become more helpful to new users of Maxima. *All* comments and suggestions for improvements will be appreciated and carefully considered.

```
LOADING FILES
The defaults allow you to use the brief version load(fft) to load in the
Maxima file fft.lisp.
To load in your own file, such as qxxx.mac
using the brief version load(qxxx), you either need to place
qxxx.mac in one of the folders Maxima searches by default, or
else put a line like:

file_search_maxima : append(["c:/work2/###.{mac,mc}"],file_search_maxima )$

in your personal startup file maxima-init.mac (see later in this chapter
for more information about this).

Otherwise you need to provide a complete path in double quotes,
as in load("c:/work2/qxxx.mac"),

We always use the brief load version in our examples, which are generated
using the XMaxima graphics interface on a Windows XP computer, and copied
into a fancy verbatim environment in a latex file which uses the fancyvrb
and color packages.
```

```
Maxima.sourceforge.net. Maxima, a Computer Algebra System. Version 5.18.1
 (2009). http://maxima.sourceforge.net/
```

The homemade function **fll(x)** (first, last, length) is used to return the first and last elements of lists (as well as the length), and is automatically loaded in with **mbe1util.mac** from Ch. 1. We will include a reference to this definition when working with lists.

This function has the definitions

```
fll(x) := [first(x),last(x),length(x)]$
declare(fll,evfun)$
```

Some of the examples used in these notes are from the Maxima html help manual or the Maxima mailing list:
http://maxima.sourceforge.net/maximalist.html.

The author would like to thank the Maxima developers for their friendly help via the Maxima mailing list.

3.1 Solving Ordinary Differential Equations

3.2 Solution of One First Order Ordinary Differential Equation (ODE)

3.2.1 Summary Table

ode2 and ic1

`gsoln : ode2 (de, u, t);`
`where de involves 'diff(u,t).`
`psoln : ic1 (gsoln, t = t0, u = u0);`

desolve

`gsoln : desolve(de, u(t) );`
`where de includes the equal sign (=)`
`and 'diff(u(t),t) and possibly u(t).`
`psoln : ratsubst(u0val,u(o),gsoln)`

plotdf

`plotdf ( dudt, [t,u], [trajectory_at, t0, u0],`
`[direction,forward], [t, tmin, tmax],`
`[u, umin, umax] )$`

rk
`points : rk ( dudt, u, u0, [t, t0, tlast, dt] )$`
`where dudt is a function of t and u which` `determines diff(u,t).`

Table 1: Methods for One First Order ODE

We will use these four different methods to solve the first order ordinary differential equation

$$\frac{d\,u}{d\,t} = e^{-t} + u \tag{3.1}$$

subject to the condition that when $t = 2$, $\quad u = -0.1$.

3.2.2 Exact Solution with ode2 and ic1

Most ordinary differential equations have no known exact solution (or the exact solution is a complicated expression involving many terms with special functions) and one normally uses approximate methods. However, some ordinary differential equations have simple exact solutions, and many of these can be found using **ode2**, **desolve**, or **contrib_ode**. The manual has the following information about **ode2**

> Function: **ode2 (eqn, dvar, ivar)**
> The function **ode2** solves an ordinary differential equation (ODE) of first or second order. It takes three arguments: an ODE given by **eqn**, the dependent variable **dvar**, and the independent variable **ivar**. When successful, it returns either an explicit or implicit solution for the dependent variable. `%c` is used to represent the integration constant in the case of first-order equations, and `%k1` and `%k2` the constants for second-order equations. The dependence of the dependent variable on the independent variable does not have to be written explicitly, as in the case of **desolve**, but the independent variable must always be given as the third argument.

If the differential equation has the structure **Left(dudt,u,t) = Right(dudt,u,t)** (here **u** is the dependent variable and **t** is the independent variable), we can always rewrite that differential equation as **de = Left(dudt,u,t) - Right(dudt,u,t) = 0**, or **de = 0**.

We can use the syntax **ode2(de,u,t)**, with the first argument an expression which includes derivatives, instead of the complete equation including the **" = 0"** on the end, and **ode2** will assume we mean **de = 0** for the differential equation. (Of course you can also use **ode2 (de=0, u, t)**

We rewrite our example linear first order differential equation Eq. 3.1 in the way just described, using the **noun** form **'diff**, which uses a single quote. We then use **ode2**, and call the general solution **gsoln**.

```
(%i1) de : 'diff(u,t)- u - exp(-t);
                                 du        - t
(%o1)                            -- - u - %e
                                 dt
(%i2) gsoln : ode2(de,u,t);
                                        - 2 t
                                      %e          t
(%o2)                         u = (%c - -------) %e
                                         2
```

The general solution returned by **ode2** contains one constant of integration `%c`, and is an explicit solution for **u** as a function of **t**, although the above does not bind the symbol `u`.

We next find the particular solution which has $t = 2,\ u = -0.1$ using **ic1**, and call this particular solution **psoln**. We then check the returned solution in two ways: 1. does it satisfy the conditions given to **ic1**?, and 2. does it produce a zero value for our expression **de**?

```
(%i3) psoln : ic1(gsoln,t = 2, u = -0.1),ratprint:false;
                           - t - 4     2         2 t       4
                         %e        ((%e  - 5) %e    + 5 %e )
(%o3)                u = - ----------------------------------
                                          10
(%i4) rhs(psoln),t=2,ratsimp;
                                       1
(%o4)                                - --
                                       10
(%i5) de,psoln,diff,ratsimp;
(%o5)                                  0
```

Both tests are passed by this particular solution. We can now make a quick plot of this solution using **plot2d**.

```
(%i6) us : rhs(psoln);
                         - t - 4     2         2 t       4
                       %e        ((%e  - 5) %e    + 5 %e )
(%o6)                - ----------------------------------
                                       10
(%i7) plot2d(us,[t,0,7],
          [style,[lines,5]],[ylabel," "],
          [xlabel,"t0 = 2, u0 = -0.1, du/dt = exp(-t) + u"])$
```

which looks like

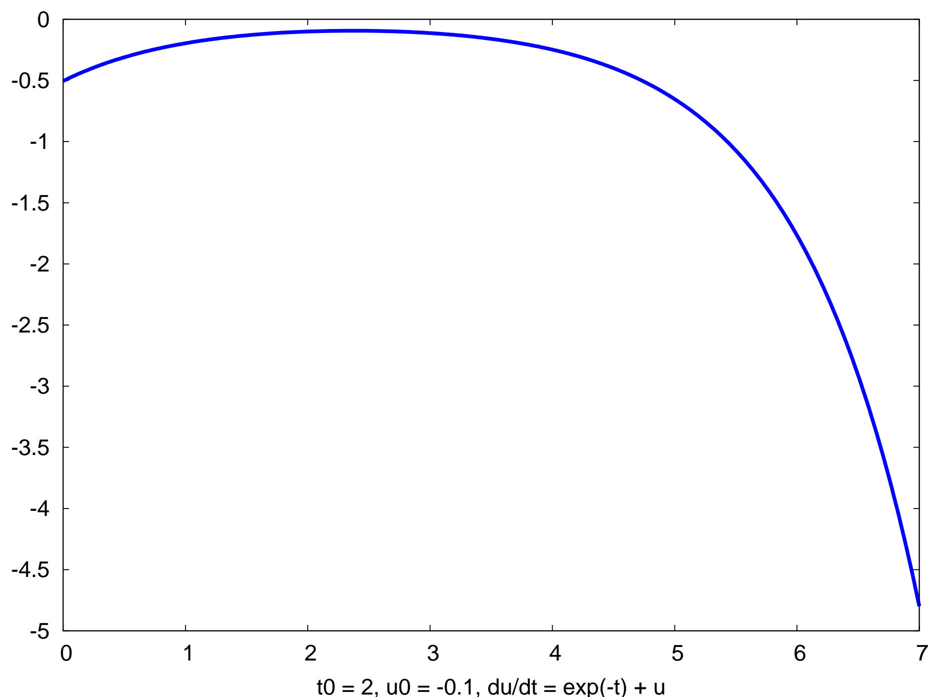

Figure 1: Solution for which $\mathbf{t = 2,\ u = -0.1}$

3.2.3 Exact Solution Using desolve

desolve uses Laplace transform methods to find an exact solution. To be able to use **desolve**, we need to write our example differential equation Eq.3.1 in a more explicit form, with every **u -> u(t)**, and include the **=** sign in the definition of the differential equation.

```
(%i1) eqn : 'diff(u(t),t) - exp(-t) - u(t) = 0;
                           d                    - t
(%o1)                      -- (u(t)) - u(t) - %e     = 0
                           dt
(%i2) gsoln : desolve(eqn,u(t));
                                         t      - t
                        (2 u(0) + 1) %e     %e
(%o2)                  u(t) = ---------------- - -----
                               2              2
(%i3) eqn,gsoln,diff,ratsimp;
(%o3)                            0 = 0
(%i4) bc : subst ( t=2, rhs(gsoln)) = - 0.1;
                          2                  - 2
                        %e   (2 u(0) + 1)    %e
(%o4)                   ---------------- - ----- = - 0.1
                               2              2
(%i5) solve ( eliminate ( [gsoln, bc],[u(0)]), u(t) ),ratprint:false;
                               - t      t - 2        t - 4
                         - 5 %e     - %e       + 5 %e
(%o5)               [u(t) = -------------------------------]
                                      10
(%i6) us : rhs(%[1]);
                            - t      t - 2        t - 4
                      - 5 %e     - %e       + 5 %e
(%o6)                 -------------------------------
                                   10
```

```
(%i7) us, t=2, ratsimp;
                                          1
(%o7)                                   - --
                                          10
(%i8) plot2d(us,[t,0,7],
           [style,[lines,5]],[ylabel," "],
           [xlabel,"t0 = 2, u0 = -0.1, du/dt = exp(-t) + u"])$
```

and we get the same plot as before. The function **desolve** returns a solution in terms of the "initial value" `u(0)`, which here means `u(t = 0)`, and we must go through an extra step to eliminate `u(0)` in a way that assures our chosen boundary condition `t = 2,  u - -0.1` is satisfied.

We have checked that the general solution satisfies the given differential equation in `%i3`, and have checked that our particular solution satisfies the desired condition at `t = 2` in `%i7`.

If your problem requires that the value of the solution `us` be specified at `t = 0`, the route to the particular solution is much simpler than what we used above. You simply use `subst ( u(0) = -1, rhs (gsoln) )` if, for example, you wanted a particular solution to have the property that when `t = 0`, `u = -1`.

```
(%i9) us : subst( u(0) = -1,rhs(gsoln) ),ratsimp;
                                    - t     2 t
                                  %e    (%e    + 1)
(%o9)                           - -----------------
                                          2
(%i10) us,t=0,ratsimp;
(%o10)                                  - 1
```

3.2.4 Numerical Solution and Plot with plotdf

We next use **plotdf** to numerically integrate the given first order ordinary differential equation, draw a direction field plot which governs any particular solution, and draw the particular solution we have chosen.

The default color choice of **plotdf** is to use small blue arrows give the local direction of the trajectory of the particular solution passing though that point. This direction can be defined by an angle α such that if $\mathbf{u}' = \mathbf{f}(\mathbf{t}, \mathbf{u})$, then $\tan(\alpha) = \mathbf{f}(\mathbf{t}, \mathbf{u})$, and at the point $(\mathbf{t_0},\ \mathbf{u_0})$,

$$\mathbf{d\,u} = \mathbf{f}(\mathbf{t_0}, \mathbf{u_0}) \times \mathbf{d\,t} = \mathbf{d\,t}\ \times \left(\frac{\mathbf{d\,u}}{\mathbf{d\,t}}\right)_{\mathbf{t}=\mathbf{t_0},\ \mathbf{u}=\mathbf{u_0}} \tag{3.2}$$

This equation determines the increase $\mathbf{d\,u}$ in the value of the dependent variable **u** induced by a small increase $\mathbf{d\,t}$ in the independent variable **t** at the point $(\mathbf{t_0},\ \mathbf{u_0})$. We then define a local vector with **t** component $\mathbf{d\,t}$ and **u** component $\mathbf{d\,u}$, and draw a small arrow in that direction at a grid of chosen points to construct a direction field associated with the given first order differential equation. The length of the small arrow can be increased some to reflect large values of the magnitude of $\mathbf{d\,u}/\mathbf{d\,t}$.

For one first order ordinary differential equation, **plotdf**, has the syntax

```
plotdf( dudt,[t,u], [trajectory_at, t0, u0], options ... )
```

in which `dudt` is the function of `(t,u)` which determines the rate of change $\mathbf{d\,u}/\mathbf{d\,t}$.

```
(%i1) plotdf(exp(-t) + u, [t, u], [trajectory_at,2,-0.1],
             [direction,forward], [t,0,7], [u, -6, 1] )$
```

produces the plot

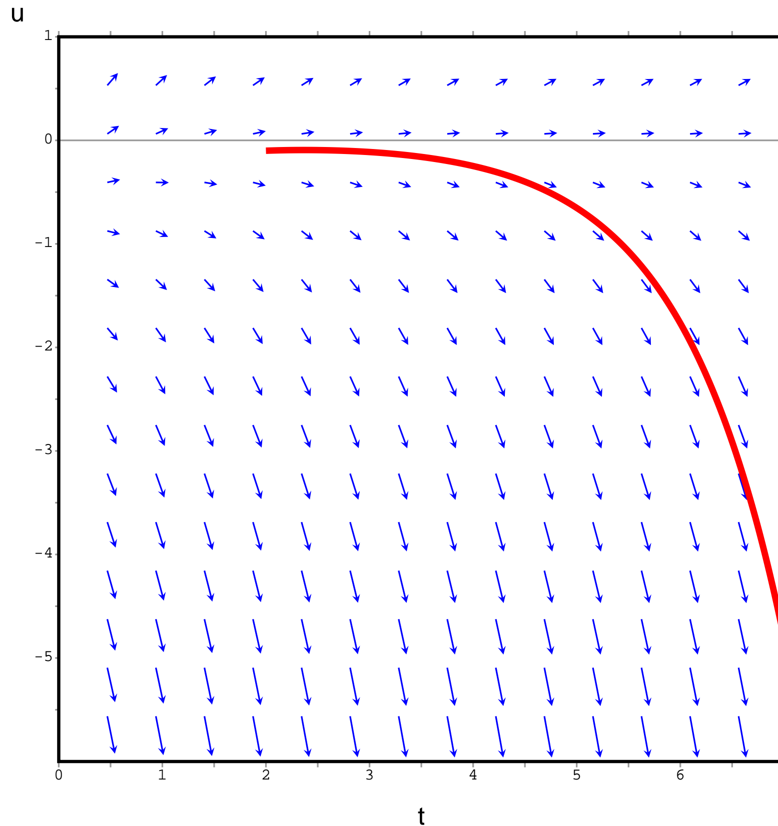

Figure 2: Direction Field for the Solution $\mathbf{t = 2,\ u = -0.1}$

(We have thickened the red curve using the **Config, Linewidth** menu option of **plotdf**, followed by **Replot**).

The help manual has an extensive discussion and examples of the use of the direction field plot utility **plotdf**.

3.2.5 Numerical Solution with 4th Order Runge-Kutta: rk

Although the **plotdf** function is useful for orientation about the shapes and types of solutions possible, if you need a list with coordinate points to use for other purposes, you can use the fourth order Runge-Kutta function **rk**.

For one first order ordinary differential equation, the syntax has some faint resemblance to that of **plotdf**:

```
rk ( dudt, u, u0, [ t, t0, tlast, dt ] )
```

in which we are assuming that **u** is the dependent variable and **t** is the independent variable, and **dudt** is that function of **(t, u)** which locally determines the value of $\mathbf{d\,u/d\,t}$. This will numerically integrate the corresponding first order ordinary differential equation and return a list of pairs of **(t, u)** on the solution curve which has been requested:

```
[ [t0, u0], [t0 + dt, y(t0 + dt)], .... ,[tlast, y(tlast)] ]
```

For our example first order ordinary differential equation, choosing the same initial conditions as above, and choosing **dt = 0.01**,

```
(%i1) fpprintprec:8$
(%i2) points : rk (exp(-t) + u, u, -0.1, [ t, 2, 7, 0.01 ] )$
(%i3) %, fll;
(%o3)                  [[2, - 0.1], [7.0, - 4.7990034], 501]
(%i4) plot2d( [ discrete, points ], [ t, 0, 7],
          [style,[lines,5]],[ylabel," "],
          [xlabel,"t0 = 2, u0 = -0.1, du/dt = exp(-t) + u"])$
```

(We have used our homemade function **fll(x)**, loaded in at startup with the other functions defined in **mbe1util.mac**, available with the Ch. 1 material. We have provided the definition of **fll** in the preface of this chapter. Instead of **%, fll ;**, you could use **[%[1],last(%),length(%)];** to get the same information.)

The plot looks like

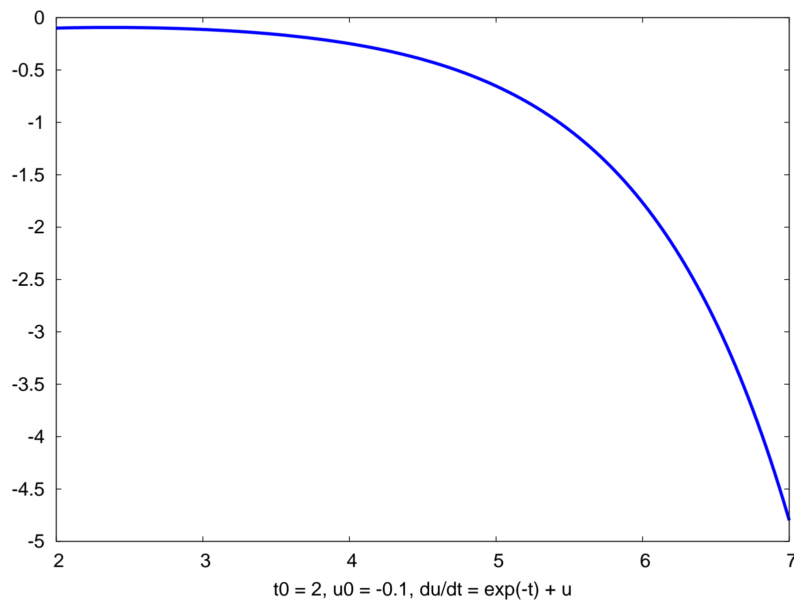

Figure 3: Runge-Kutta Solution with $\mathbf{t = 2,\ u = -0.1}$

3.3 Solution of One Second Order ODE or Two First Order ODE's

3.3.1 Summary Table

ode2 and ic1
gsoln : ode2 (de, u, t); where de involves 'diff(u,t,2) and possibly 'diff(u,t). psoln : ic2 (gsoln, t = t0, u = u0, 'diff(u,t) = up0);
desolve

atvalue ('diff(u,t), t = 0, v(0));
gsoln : desolve(de, u(t));
where de includes the equal sign (=), 'diff(u(t),t,2),
and possibly 'diff(u(t),t) and u(t).
One type of particular solution is returned by using
psoln : subst([u(o) = u0, v(0) = v0] , gsoln)

plotdf
plotdf ([dudt, dvdt], [u, v], [trajectory_at, u0, v0], [u, umin, umax],[v, vmin, vmax], [tinitial, t0], [direction,forward], [versus_t, 1],[tstep, timestepval], [nsteps, nstepsvalue])$
rk
points : rk ([dudt, dvdt],[u, v],[u0, v0],[t, t0, tlast, dt])$
where dudt and dvdt are functions of t,u, and v which determine diff(u,t) and diff(v,t).

Table 2: Methods for One Second Order or Two First Order ODE's

We apply the above four methods to the simple second order ordinary differential equation:

$$\frac{d^2 u}{d t^2} = 4 u \tag{3.3}$$

subject to the conditions that when $t = 2$, $u = 1$ and $d u/d t = 0$.

3.3.2 Exact Solution with ode2, ic2, and eliminate

The main difference here is the use of **ic2** rather than **ic1**.

```
(%i1) de : 'diff(u,t,2) - 4*u;
                                   2
                                  d u
(%o1)                             --- - 4 u
                                    2
                                  dt
(%i2) gsoln : ode2(de,u,t);
                                   2 t          - 2 t
(%o2)                     u = %k1 %e    + %k2 %e
(%i3) de,gsoln,diff,ratsimp;
(%o3)                                0
```

```
(%i4) psoln : ic2(gsoln,t=2,u=1,'diff(u,t) = 0);
                                  2 t - 4     4 - 2 t
                                %e          %e
(%o4)                       u = --------- + ---------
                                    2           2
(%i5) us : rhs(psoln);
                                2 t - 4     4 - 2 t
                              %e          %e
(%o5)                         --------- + ---------
                                  2           2
(%i6) us, t=2, ratsimp;
(%o6)                                  1
(%i7) plot2d(us,[t,0,4],[y,0,10],
           [style,[lines,5]],[ylabel," "],
           [xlabel," U versus t, U''(t) = 4 U(t), U(2) = 1, U'(2) = 0 "])$
plot2d: expression evaluates to non-numeric value somewhere in plotting range.
```

which produces the plot

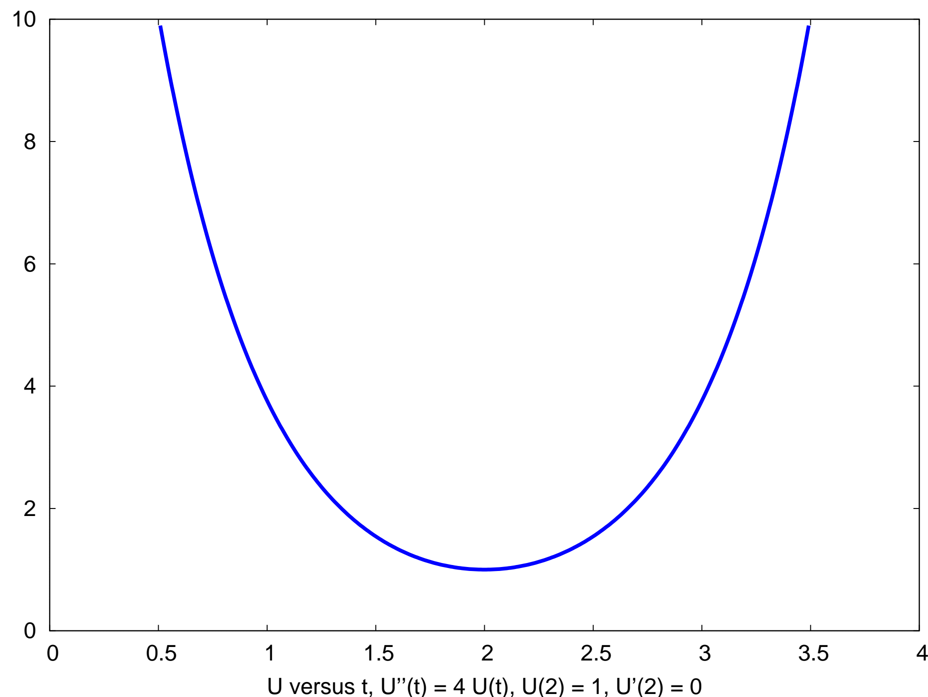

Figure 4: Solution for which $\mathbf{t = 2,\ u = 1,\ u' = 0}$

Next we make a "phase space plot" which is a plot of $\mathbf{v = d\,u/d\,t}$ versus $\mathbf{u}$ over the range $\mathbf{1 \le t \le 3}$.

```
(%i8) vs : diff(us,t),ratsimp;
                              - 2 t - 4     4 t      8
(%o8)                       %e          (%e     - %e )
(%i9) for i thru 3 do
        d[i]:[discrete,[float(subst(t=i,[us,vs]))]]$
(%i10) plot2d( [[parametric,us,vs,[t,1,3]],d[1],d[2],d[3] ],
            [x,0,8],[y,-12,12],
          [style, [lines,5,1],[points,4,2,1],
            [points,4,3,1],[points,4,6,1]],
          [ylabel," "],[xlabel," "],
           [legend," du/dt vs u "," t = 1 ","t = 2","t = 3"] )$
```

which produces the plot

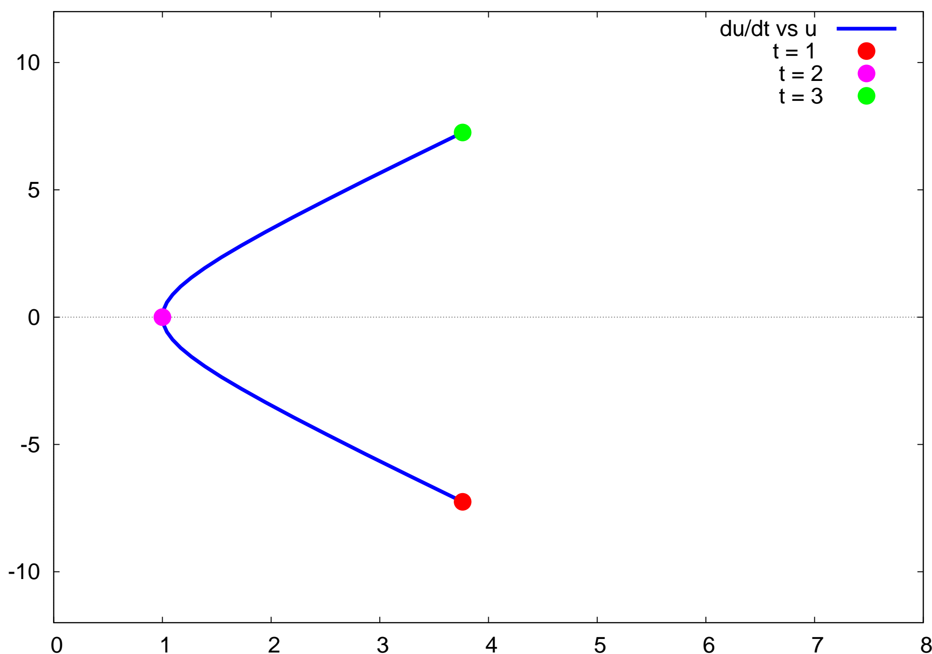

Figure 5: $\mathbf{t = 2,\ y = 1,\ y' = 0}$ Solution

If your boundary conditions are, instead, for `t=0, u = 1`, and for `t = 2, u = 4`, then one can eliminate the two constants "by hand" instead of using **ic2** (see also next section).

```
(%i4) bc1 : subst(t=0,rhs(gsoln)) = 1$
(%i5) bc2 : subst(t = 2, rhs(gsoln)) = 4$
(%i6) solve(
        eliminate([gsoln,bc1,bc2],[%k1,%k2]), u ),
           ratsimp, ratprint:false;
                        - 2 t       4         4 t     8       4
                      %e      ((4 %e  - 1) %e    + %e  - 4 %e )
(%o6)            [u = ----------------------------------------]
                                         8
                                       %e  - 1
(%i7) us : rhs(%[1]);
                      - 2 t       4         4 t     8       4
                    %e      ((4 %e  - 1) %e    + %e  - 4 %e )
(%o7)               ----------------------------------------
                                       8
                                     %e  - 1
(%i8) us,t=0,ratsimp;
(%o8)                                  1
(%i9) us,t=2,ratsimp;
(%o9)                                  4
```

3.3.3 Exact Solution with desolve, atvalue, and eliminate

The function **desolve** uses Laplace transform methods which are set up to expect the use of initial values for dependent variables and their derivatives. (However, we will show how you can impose more general boundary conditions.) If the dependent variable is **u(t)**, for example, the solution is returned in terms of a constant **u(0)**, which refers to the value of **u(t = 0)** (here we are assuming that the independent variable is **t**). To get a simple result from **desolve** which we can work with (for the case of a second order ordinary differential equation), we can use the **atvalue** function with the syntax (for example):

```
atvalue ( 'diff(u,t), t = 0, v(0) )
```

which will allow **desolve** to return the solution to a second order ODE in terms of the pair of constants **(u(0), v(0))**. Of course, there is nothing sacred about using the symbol **v(0)** here. The function **atvalue** should be invoked before the use of **desolve**.

If the desired boundary conditions for a particular solution refer to **t = 0**, then you can immediately find that particular solution using the syntax (if **ug** is the general solution, say)

```
us : subst( [ u(0) = u0val, v(0) = v0val], ug ),
```

or else by using **ratsubst** twice.

In our present example, the desired boundary conditions refer to **t = 2** , and impose conditions on the value of **u** and its first derivative at that value of **t**. This requires a little more work, and we use **eliminate** to get rid of the constants **(u(0), v(0))** in a way that allows our desired conditions to be satisfied.

```
(%i1) eqn : 'diff(u(t),t,2) - 4*u(t) = 0;
                                2
                               d
(%o1)                          --- (u(t)) - 4 u(t) = 0
                                 2
                               dt
(%i2) atvalue ( 'diff(u(t),t), t=0, v(0) )$
(%i3) gsoln : desolve(eqn,u(t));
                                         2 t                       - 2 t
                 (v(0) + 2 u(0)) %e       (v(0) - 2 u(0)) %e
(%o3)       u(t) = --------------------- - ----------------------
                                 4                        4
(%i4) eqn,gsoln,diff,ratsimp;
(%o4)                                    0 = 0
(%i5) ug : rhs(gsoln);
                                      2 t                      - 2 t
              (v(0) + 2 u(0)) %e       (v(0) - 2 u(0)) %e
(%o5)         --------------------- - ----------------------
                              4                        4
(%i6) vg : diff(ug,t),ratsimp$
(%i7) ubc : subst(t = 2,ug) = 1$
(%i8) vbc : subst(t = 2,vg) = 0$
(%i9) solve (
        eliminate([gsoln, ubc, vbc],[u(0), v(0)]), u(t) ),
         ratsimp,ratprint:false;
                                  - 2 t - 4    4 t     8
                                %e           (%e    + %e )
(%o9)                   [u(t) = -------------------------]
                                             2
```

```
(%i10) us : rhs(%[1]);
                              - 2 t - 4    4 t     8
                            %e          (%e    + %e )
(%o10)                      -------------------------
                                        2
(%i11) subst(t=2, us),ratsimp;
(%o11)                                  1
(%i12) vs : diff(us,t),ratsimp;
                              - 2 t - 4    4 t     8
(%o12)                      %e          (%e    - %e )
(%i13) subst(t = 2,vs),ratsimp;
(%o13)                                  0
(%i14) plot2d(us,[t,0,4],[y,0,10],
           [style,[lines,5]],[ylabel," "],
           [xlabel," U versus t, U''(t) = 4 U(t), U(2) = 1, U'(2) = 0 "])$
plot2d: expression evaluates to non-numeric value somewhere in plotting range.
(%i15) for i thru 3 do
        d[i]:[discrete,[float(subst(t=i,[us,vs]))]]$
(%i16) plot2d( [[parametric,us,vs,[t,1,3]],d[1],d[2],d[3] ],
            [x,0,8],[y,-12,12],
          [style, [lines,5,1],[points,4,2,1],
            [points,4,3,1],[points,4,6,1]],
          [ylabel," "],[xlabel," "],
           [legend," du/dt vs u "," t = 1 ","t = 2","t = 3"] )$
```

which generates the same plots found with the **ode2** method above.

If the desired boundary conditions are that **u** have given values at **t = 0** and **t = 3**, then we can proceed from the same general solution above as follows with **up** being a partially defined particular solution (assume **u(0) = 1** and **u(3) = 2**):

```
(%i17) up : subst(u(0) = 1, ug);
                                     2 t                  - 2 t
                     (v(0) + 2) %e       (v(0) - 2) %e
(%o17)               ---------------- - ------------------
                            4                    4
(%i18) ubc : subst ( t=3, up) = 2;
                       6                 - 6
                     %e  (v(0) + 2)    %e    (v(0) - 2)
(%o18)               -------------- - ---------------- = 2
                           4                 4
(%i19) solve(
          eliminate ( [ u(t) = up, ubc ],[v(0)] ), u(t) ),
             ratsimp, ratprint:false;
                         - 2 t       6        4 t     12       6
                       %e      ((2 %e  - 1) %e    + %e   - 2 %e )
(%o19)         [u(t) = ------------------------------------------]
                                          12
                                        %e   - 1
(%i20) us : rhs (%[1]);
                     - 2 t       6        4 t     12       6
                   %e      ((2 %e  - 1) %e    + %e   - 2 %e )
(%o20)             ------------------------------------------
                                      12
                                    %e   - 1
(%i21) subst(t = 0, us),ratsimp;
(%o21)                                  1
(%i22) subst (t = 3, us),ratsimp;
(%o22)                                  2
```

```
(%i23) plot2d(us,[t,0,4],[y,0,10],
           [style,[lines,5]],[ylabel," "],
           [xlabel," U versus t, U'' (t) = 4 U(t), U(0) = 1, U(3) = 2 "])$
plot2d: expression evaluates to non-numeric value somewhere in plotting range.
```

which produces the plot

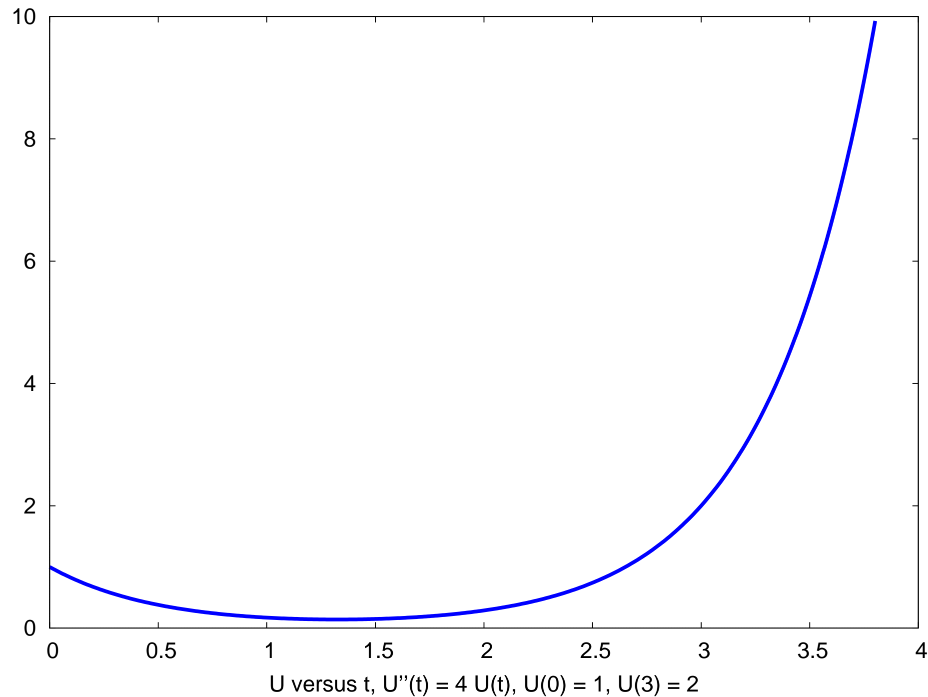

Figure 6: Solution for $\mathbf{u(0) = 1,\ u(3) = 2}$

If instead, you need to satisfy **u(1) = -1** and **u(3) = 2**, you could proceed from **gsoln** and **ug** as follows:

```
(%i24) ubc1 : subst ( t=1, ug) = -1$
(%i25) ubc2 : subst ( t=3, ug) = 2$
(%i26) solve(
         eliminate ( [gsoln, ubc1, ubc2],[u(0),v(0)]), u(t) ),
          ratsimp, ratprint:false;
                        - 2 t        4         4 t      12        8
                     %e      ((2 %e  + 1) %e     - %e    - 2 %e )
(%o26)        [u(t) = ------------------------------------------]
                                         10      2
                                       %e   - %e
(%i27) us : rhs(%[1]);
                 - 2 t        4         4 t      12        8
              %e      ((2 %e  + 1) %e     - %e    - 2 %e )
(%o27)        ------------------------------------------
                                  10      2
                                %e   - %e
(%i28) subst ( t=1, us), ratsimp;
(%o28)                               - 1
(%i29) subst ( t=3, us), ratsimp;
(%o29)                               2
(%i30) plot2d ( us, [t,0,4], [y,-2,8],
          [style,[lines,5]],[ylabel," "],
          [xlabel," U versus t, U'' (t) = 4 U(t), U(1) = -1, U(3) = 2 "])$
```

which produces the plot

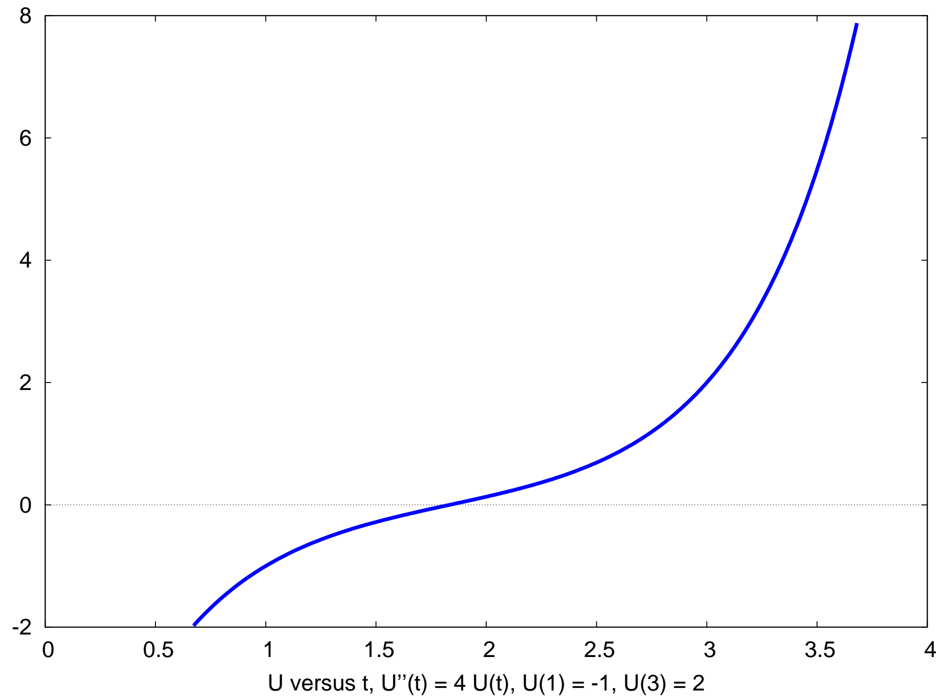

Figure 7: Solution for $\mathbf{u(1) = -1,\ u(3) = 2}$

The simplest case of using **desolve** is the case in which you impose conditions on the solution and its first derivative at `t = 0`, in which case you simply use:

```
(%i4) psoln : subst([u(0) = 1,v(0)=0],gsoln);
                                         2 t      - 2 t
                                       %e       %e
(%o4)                         u(t) = ----- + -------
                                         2         2
(%i5) us : rhs(psoln);
                                      2 t      - 2 t
                                    %e       %e
(%o5)                               ----- + -------
                                      2         2
```

in which we have chosen the initial conditions `u(0) = 1`, and `v(0) = 0`.

3.3.4 Numerical Solution and Plot with plotdf

Given a second order autonomous ODE, one needs to introduce a second dependent variable **v(t)**, say, which is defined as the first derivative of the original single dependent variable **u(t)**. Then for our example, the starting ODE

$$\frac{d^2 u}{d t^2} = 4 u \tag{3.4}$$

is converted into two first order ODE's

$$\frac{d u}{d t} = v, \quad \frac{d v}{d t} = 4 u \tag{3.5}$$

and the **plotdf** syntax for two first order ODE's is

```
plotdf ( [dudt, dvdt], [u, v], [trajectory_at, u0, v0], [u, umin, umax],
         [v, vmin, vmax], [tinitial, t0], [versus_t, 1],
         [tstep, timestepval], [nsteps, nstepsvalue] )$
```

in which at **t = t0**, **u = u0** and **v = v0**. If **t0 = 0** you can omit the option **[tinitial, t0]**. The options **[u, umin, umax]** and **[v, vmin, vmax]** allow you to control the horizontal and vertical extent of the phase space plot (here **v** versus **u**) which will be produced. The option **[versus_t,1]** tells **plotdf** to create a separate plot of both **u** and **v** versus the dependent variable. The last two options are only needed if you are not satisfied with the plots and want to experiment with other than the default values of **tstep** and **nsteps**.

Another option you can add is **[direction,forward]**, which will display the trajectory for **t** greater than or equal to **t0**, rather than for a default interval around the value **t0** which corresponds to **[direction,both]**.
Here we invoke **plotdf** for our example.

```
(%i1) plotdf ( [v, 4*u], [u, v], [trajectory_at, 1, 0],
               [u, 0, 8], [v, -10, 10], [versus_t, 1],
                 [tinitial, 2])$
```

The plot versus **t** is

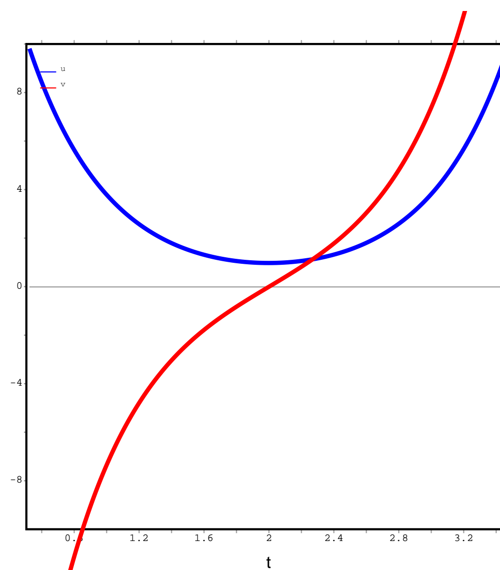

Figure 8: u(t) and u'(t) vs. $\mathbf{t}$ for $\mathbf{u(2) = 1,\ u'(2) = 0}$

and the phase space plot is

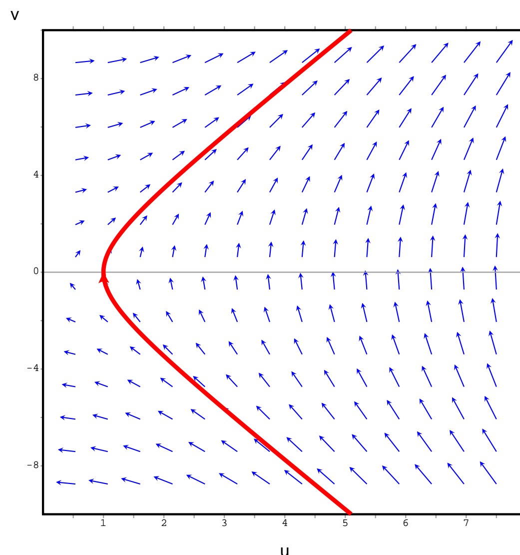

Figure 9: u'(t) vs. u(t) for $\mathbf{u(2) = 1,\ u'(2) = 0}$

In both of these plots we used the **`Config`** menu to increase the linewidth, and then clicked on **`Replot`**. We also cut and pasted the color **`red`** to be the second choice on the color cycle (instead of green) used in the plot versus the independent variable **`t`**. Note that no matter what you call your independent variable, it will always be called **`t`** on the plot of the dependent variables versus the independent variable.

3.3.5 Numerical Solution with 4th Order Runge-Kutta: rk

To use the fourth order Runge-Kutta numerical integrator **rk** for this example, we need to follow the procedure used in the previous section using **plotdf**, converting the second order ODE to a pair of first order ODE's.

The syntax for two first order ODE's with dependent variables **`[u,v]`** and independent variable **`t`** is

```
rk ( [ dudt, dvdt ], [u,v], [u0,v0], [t, t0, tmax, dt] )
```

which will produce the list of lists:

```
[ [t0, u0,v0],[t0+dt, u(t0+dt),v(t0+dt)], ..., [tmax, u(tmax),v(tmax)] ]
```

For our example, following our discussion in the previous section with **plotdf**, we use

```
points : rk ( [v, 4*u], [u, v], [1, 0], [t, 2, 3.6, 0.01] )
```

We again use the homemade function **`fll`** (see the preface) to look at the first element, the last element, and the length of various lists.

```
(%i1) fpprintprec:8$
(%i2) points : rk([v,4*u],[u,v],[1,0],[t,2,3.6,0.01])$
(%i3) %, fll;
(%o3)           [[2, 1, 0], [3.6, 12.286646, 24.491768], 161]
(%i4) uL : makelist([points[i][1],points[i][2]],i,1,length(points))$
(%i5) %, fll;
(%o5)                  [[2, 1], [3.6, 12.286646], 161]
```

```
(%i6) vL : makelist([points[i][1],points[i][3]],i,1,length(points))$
(%i7) %, fll;
(%o7)                    [[2, 0], [3.6, 24.491768], 161]
(%i8) plot2d([ [discrete,uL],[discrete,vL]],[x,1,5],
           [style,[lines,5]],[y,-1,24],[ylabel," "],
           [xlabel,"t"],[legend,"u(t)","v(t)"])$
```

which produces the plot

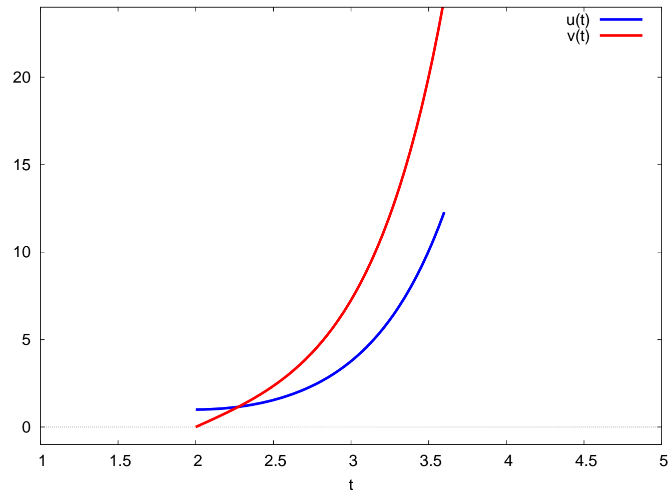

Figure 10: Runge-Kutta for $\mathbf{u(2) = 1,\ u'(2) = 0}$

Next we make a phase space plot of $\mathbf{v}$ versus $\mathbf{u}$ from the result of the Runge-Kutta integration.

```
(%i9) uvL : makelist([points[i][2],points[i][3]],i,1,length(points))$
(%i10) %, fll;
(%o10)                 [[1, 0], [12.286646, 24.491768], 161]
(%i11) plot2d( [ [discrete,uvL]],[x,0,13],[y,-1,25],
                [style,[lines,5]],[ylabel," "],
                [xlabel," v vs. u "])$
```

which produces the phase space plot

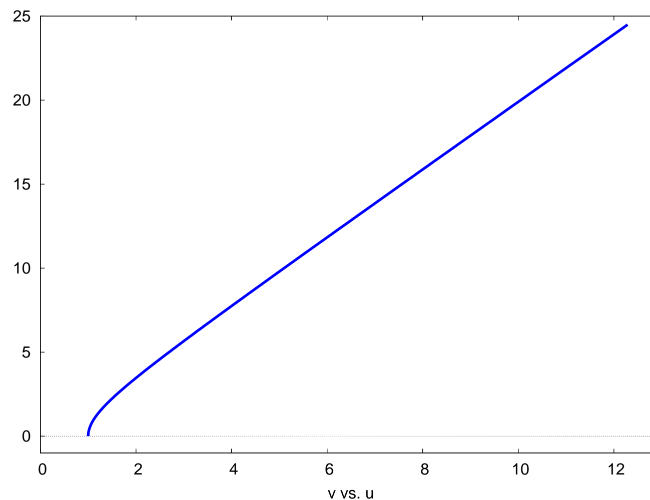

Figure 11: R-K Phase Space Plot for $\mathbf{u(2) = 1,\ u'(2) = 0}$

3.4 Examples of ODE Solutions

3.4.1 Ex. 1: Fall in Gravity with Air Friction: Terminal Velocity

Let's explore a problem posed by Patrick T. Tam (A Physicist's Guide to Mathematica, Academic Press, 1997, page 349).

> A small body falls downward with an initial velocity $\mathbf{v_0}$ from a height $\mathbf{h}$ near the surface of the earth. For low velocities (less than about $\mathbf{24\,m/s}$), the effect of air resistance may be approximated by a frictional force proportional to the velocity. Find the displacement and velocity of the body, and determine the terminal velocity. Plot the speed as a function of time for several initial velocities.

The net vector force $\mathbf{F}$ acting on the object is thus assumed to be the (constant) force of gravity and the (variable) force due to air friction, which is in a direction opposite to the direction of the velocity vector $\mathbf{v}$. We can then write Newton's Law of motion in the form

$$\mathbf{F} = m\,\mathbf{g} - b\,\mathbf{v} = m\,\frac{d\,\mathbf{v}}{d\,t} \tag{3.6}$$

In this vector equation, m is the mass in kg., $\mathbf{g}$ is a vector pointing downward with magnitude g, and b is a positive constant which depends on the size and shape of the object and on the viscosity of the air. The velocity vector $\mathbf{v}$ points down during the fall.

If we choose the y axis positive downward, with the point $y = 0$ the launch point, then the net y components of the force and Newton's Law of motion are:

$$F_y = m\,g - b\,v_y = m\,\frac{d\,v_y}{d\,t} \tag{3.7}$$

where g is the positive number $9.8\,m/s^2$ and since the velocity component $v_y > 0$ during the fall, the effects of gravity and air resistance are in competition.

We see that the rate of change of velocity will become zero at the instant that $m\,g - b\,v_y = 0$, or $v_y = m\,g/b$, and the downward velocity stops increasing at that moment, the "**terminal velocity**" having been attained.

While working with Maxima, we can simplify our notation and let $v_y \to v$ and $(b/m) \to a$ so both v and a represent positive numbers. We then use Maxima to solve the equation $d\,v/d\,t = g - a\,v$. The dimension of each term of this equation must evidently be the dimension of v/t, so a has dimension $1/t$.

```
(%i1) de : 'diff(v,t) - g + a*v;
                                dv
(%o1)                           -- + a v - g
                                dt
(%i2) gsoln : ode2(de,v,t);
                                          a t
                               - a t  g %e
(%o2)                     v = %e      (------- + %c)
                                         a
(%i3) de, gsoln, diff,ratsimp;
(%o3)                                 0
```

We then use **ic1** to get a solution such that $v = v0$ when $t = 0$.

```
(%i4) psoln : expand ( ic1 (gsoln,t = 0, v = v0 ) );
                                             - a t
                             - a t       g %e         g
(%o4)                   v = %e      v0 - --------- + -
                                             a        a
(%i5) vs : rhs(psoln);
                                          - a t
                           - a t      g %e         g
(%o5)                    %e      v0 - --------- + -
                                          a        a
```

For consistency, we must get the correct **terminal speed** for large t:

```
(%i6) assume(a>0)$
(%i7) limit( vs, t, inf );
                                        g
(%o7)                                   -
                                        a
```

which agrees with our analysis.

To make some plots, we can introduce a dimensionless time u with the replacement $t \rightarrow u = a\,t$, and a dimensionless speed w with the replacement $v \rightarrow w = a\,v/g$.

```
(%i8) expand(vs*a/g);
                                - a t
                            a %e      v0      - a t
(%o8)                       ------------ - %e      + 1
                                 g
(%i9) %,[t=u/a,v0=w0*g/a];
                                 - u         - u
(%o9)                          %e    w0 - %e    + 1
(%i10) ws : collectterms (%, exp (-u));
                                   - u
(%o10)                           %e    (w0 - 1) + 1
```

As our dimensionless time u gets large, $\mathbf{ws} \rightarrow \mathbf{1}$, which is the value of the terminal speed in dimensionless units.

Let's now plot three cases, two cases with initial speed less than terminal speed and one case with initial speed greater than the terminal speed. (The use of dimensionless units for plots generates what are called "universal curves", since they are generally valid, no matter what the actual numbers are).

```
(%i11) plot2d([[discrete,[[0,1],[5,1]]],subst(w0=0,ws),subst(w0=0.6,ws),
                    subst(w0=1.5,ws)],[u,0,5],[y,0,2],
           [style,[lines,2,7],[lines,4,1],[lines,4,2],[lines,4,3]],
           [legend,"terminal speed", "w0 = 0", "w0 = 0.6", "w0 = 1.5"],
         [ylabel, " "],
         [xlabel, " dimensionless speed w vs dimensionless time u"])$
```

which produces:

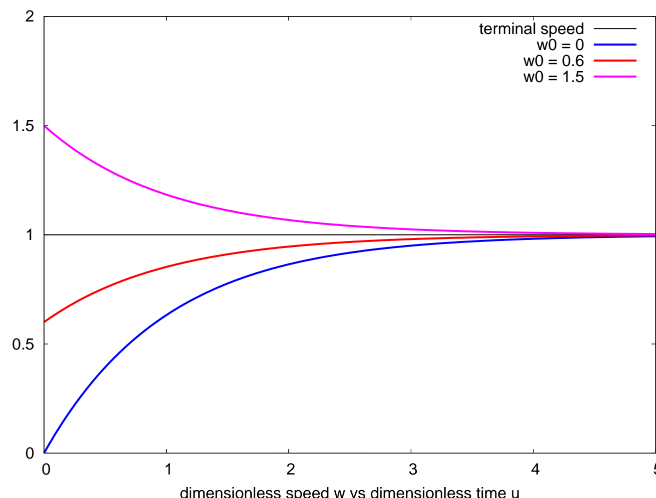

Figure 12: Dimensionless Speed Versus Dimensionless Time

An object thrown down with an initial speed greater than the terminal speed (as in the top curve) slows down until its speed is the terminal speed.

Thus far we have been only concerned with the relation between velocity and time. We can now focus on the implications for distance versus time. A dimensionless length $\mathbf{z}$ is $\mathbf{a^2\,y/g}$ and the relation $\mathbf{d\,y/d\,t = v}$ becomes $\mathbf{d\,z/d\,u = w}$, or $\mathbf{d\,z = w\,d\,u}$, which can be integrated over corresponding intervals: $\mathbf{z}$ over the interval $[\mathbf{0}, \mathbf{z_f}]$, and $\mathbf{u}$ over the interval $[\mathbf{0}, \mathbf{u_f}]$.

```
(%i12) integrate(1,z,0,zf) = integrate(ws,u,0,uf);
                                - uf            uf
(%o12)            zf = - %e       (w0 - uf %e    - 1) + w0 - 1
(%i13) zs : expand(rhs(%)),uf = u;
                                - u             - u
(%o13)                      - %e    w0 + w0 + %e    + u - 1
(%i14) zs, u=0;
(%o14)                                  0
```

(Remember the object is launched at $\mathbf{y = 0}$ which means at $\mathbf{z = 0}$). Let's make a plot of distance travelled vs time (dimensionless units) for the three cases considered above.

```
(%i15) plot2d([subst(w0=0,zs),subst(w0=0.6,zs),
          subst(w0=1.5,zs)],[u,0,1],[style,[lines,4,1],[lines,4,2],
                    [lines,4,3]], [legend,"w0 = 0", "w0 = 0.6", "w0 = 1.5"],
          [ylabel," "],
          [xlabel,"dimensionless distance z vs dimensionless time u"],
          [gnuplot_preamble,"set key top left;"])$
```

which produces:

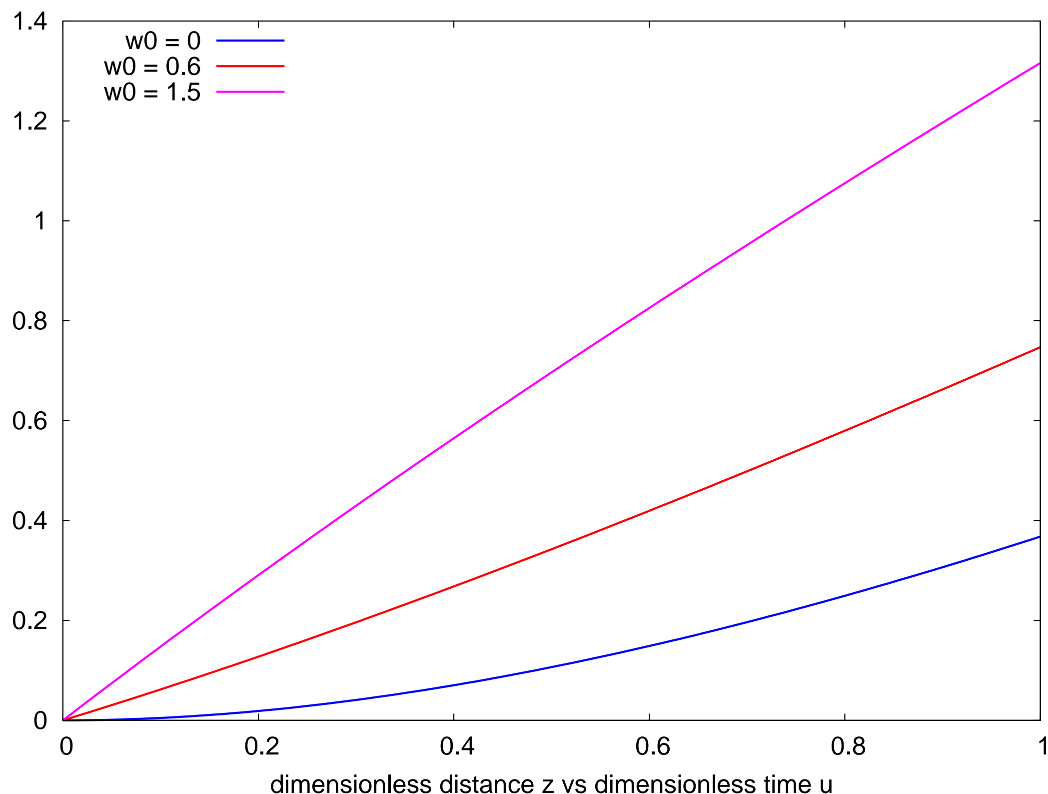

Figure 13: Dimensionless Distance Versus Dimensionless Time

3.4.2 Ex. 2: One Nonlinear First Order ODE

Let's solve

$$x^2\, y\, \frac{d\, y}{d\, x} = x\, y^2 + x^3 - 1 \tag{3.8}$$

for a solution such that when $x = 1,\ y = 1$.

```
(%i1) de : x^2*y*'diff(y,x) - x*y^2 - x^3 + 1;
                              2   dy       2    3
(%o1)                        x  y -- - x y  - x  + 1
                                  dx
(%i2) gsoln : ode2(de,y,x);
                               2      3
                        3 x y  - 6 x  log(x) - 2
(%o2)                   ------------------------ = %c
                                    3
                                 6 x
(%i3) psoln : ic1(gsoln,x=1,y=1);
                               2      3
                        3 x y  - 6 x  log(x) - 2   1
(%o3)                   ------------------------ = -
                                    3              6
                                 6 x
```

This implicitly determines y as a function of the independent variable x By inspection, we see that $x = 0$ is a singular point we should stay away from, so we assume from now on that $x \neq 0$.

To look at explicit solutions **y(x)** we use **solve**, which returns a list of two expressions depending on **x**. Since the implicit solution is a quadratic in **y**, we will get two solutions from **solve**, which we call **y1** and **y2**.

```
(%i4) [y1,y2] : map('rhs, solve(psoln,y) );
                    2            2   2            2            2   2
             sqrt(6 x  log(x) + x  + -)   sqrt(6 x  log(x) + x  + -)
                                     x                            x
(%o4)     [- --------------------------, --------------------------]
                      sqrt(3)                     sqrt(3)
(%i5) [y1,y2], x = 1, ratsimp;
(%o5)                             [- 1, 1]
(%i6) de, diff, y= y2, ratsimp;
(%o6)                                0
```

We see from the values at $x = 1$ that $y2$ is the particular solution we are looking for, and we have checked that $y2$ satisfies the original differential equation. From this example, we learn the lesson that **ic1** sometimes needs some help in finding the particular solution we are looking for.

Let's make a plot of the two solutions found.

```
(%i7) plot2d([y1,y2],[x,0.01,5],
        [style,[lines,5]],[ylabel, " Y "],
          [xlabel," X "] , [legend,"Y1", "Y2"],
         [gnuplot_preamble,"set key bottom center;"])$
```

which produces:

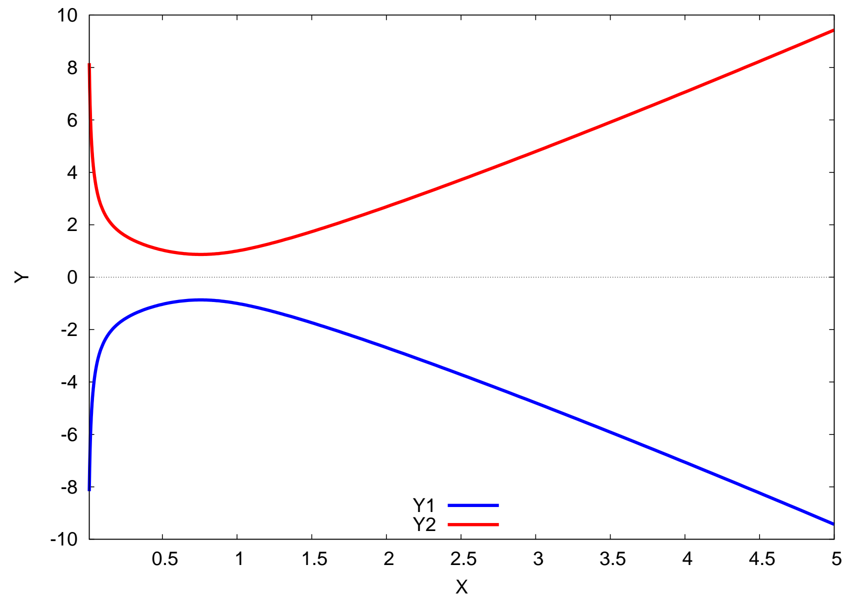

Figure 14: Positive X Solutions

3.4.3 Ex. 3: One First Order ODE Which is Not Linear in Y'

The differential equation to solve is

$$\left(\frac{d\,x}{d\,t}\right)^2 + 5\,x^2 = 8 \tag{3.9}$$

with the initial conditions $t = 0, \quad x = 0$.

```
(%i1) de: 'diff(x,t)^2 + 5*x^2 - 8;
                                  dx 2       2
(%o1)                            (--)  + 5 x  - 8
                                  dt
(%i2) ode2(de,x,t);
                                  dx 2       2
(%t2)                            (--)  + 5 x  - 8
                                  dt

                     first order equation not linear in y'

(%o2)                                   false
```

We see that direct use of **ode2** does not succeed. We can use **solve** to get equations which are linear in the first derivative, and then using **ode2** on each of the resulting linear ODE's.

```
(%i3) solve(de,'diff(x,t));
                  dx                 2    dx                 2
(%o3)            [-- = - sqrt(8 - 5 x ),  -- = sqrt(8 - 5 x )]
                  dt                      dt
(%i4) ode2 ( %[2], x, t );
                                    5 x
                          asin(----------)
                               2 sqrt(10)
(%o4)                     ---------------- = t + %c
                              sqrt(5)
```

```
(%i5) solve(%,x);
                         2 sqrt(10) sin(sqrt(5) t + sqrt(5) %c)
(%o5)               [x = --------------------------------------]
                                           5
(%i6) gsoln2 : %[1];
                         2 sqrt(10) sin(sqrt(5) t + sqrt(5) %c)
(%o6)                x = --------------------------------------
                                           5
(%i7) trigsimp ( ev (de,gsoln2,diff ) );
(%o7)                                     0
(%i8) psoln : ic1 (gsoln2, t=0, x=0);
solve: using arc-trig functions to get a solution.
Some solutions will be lost.
                                2 sqrt(10) sin(sqrt(5) t)
(%o8)                       x = -------------------------
                                            5
(%i9) xs : rhs(psoln);
                              2 sqrt(10) sin(sqrt(5) t)
(%o9)                         -------------------------
                                          5
(%i10) xs,t=0;
(%o10)                                     0
```

We have selected only one of the linear ODE's to concentrate on here. We have shown that the solution satisfies the original differential equation and the given boundary condition.

3.4.4 Ex. 4: Linear Oscillator with Damping

The equation of motion for a particle of mass **m** executing one dimensional motion which is subject to a linear restoring force proportional to `|x|` and subject to a frictional force proportional to its speed is

$$\mathbf{m}\,\frac{\mathbf{d^2\,x}}{\mathbf{d\,t^2}} + \mathbf{b}\,\frac{\mathbf{d\,x}}{\mathbf{d\,t}} + \mathbf{k\,x} = \mathbf{0} \tag{3.10}$$

Dividing by the mass **m**, we note that if there were no damping, this motion would reduce to a linear oscillator with the angular frequency

$$\omega_0 = \left(\frac{\mathbf{k}}{\mathbf{m}}\right)^{1/2}. \tag{3.11}$$

In the presence of damping, we can define

$$\gamma = \frac{\mathbf{b}}{\mathbf{2\,m}} \tag{3.12}$$

and the equation of motion becomes

$$\frac{\mathbf{d^2\,x}}{\mathbf{d\,t^2}} + \mathbf{2}\,\gamma\,\frac{\mathbf{d\,x}}{\mathbf{d\,t}} + \omega_0^2\,\mathbf{x} = \mathbf{0} \tag{3.13}$$

In the presence of damping, there are now two natural time scales

$$\mathbf{t1} = \frac{\mathbf{1}}{\omega_0}, \qquad \mathbf{t2} = \frac{\mathbf{1}}{\gamma} \tag{3.14}$$

and we can introduce a dimensionless time $\theta = \omega_0\,\mathbf{t}$ and the dimensionless positive constant $\mathbf{a} = \gamma/\omega_0$, to get

$$\frac{\mathbf{d^2\,x}}{\mathbf{d\,\theta^2}} + \mathbf{2\,a}\,\frac{\mathbf{d\,x}}{\mathbf{d\,\theta}} + \mathbf{x} = \mathbf{0} \tag{3.15}$$

The "underdamped" case corresponds to $\gamma < \omega_0$, or $\mathbf{a} < \mathbf{1}$ and results in damped oscillations around the final $\mathbf{x} = \mathbf{0}$. The "critically damped" case corresponds to $\mathbf{a} = \mathbf{1}$, and the "overdamped" case corresponds to $\mathbf{a} > \mathbf{1}$. We specialize to solutions which have the initial conditions $\boldsymbol{\theta} = \mathbf{0}, \quad \mathbf{x} = \mathbf{1}, \quad \mathbf{d\,x/d\,t} = \mathbf{0} \;\Rightarrow \mathbf{d\,x/d}\,\boldsymbol{\theta} = \mathbf{0}$.

```
(%i1) de : 'diff(x,th,2) + 2*a*'diff(x,th) + x ;
                                 2
                                d x        dx
(%o1)                           ---- + 2 a --- + x
                                  2        dth
                                dth
(%i2) for i thru 3 do
          x[i] : rhs ( ic2 (ode2 (subst(a=i/2,de),x,th), th=0,x=1,diff(x,th)=0))$
(%i3) plot2d([x[1],x[2],x[3]],[th,0,10],
          [style,[lines,4]],[ylabel," "],
           [xlabel," Damped Linear Oscillator " ],
           [gnuplot_preamble,"set zeroaxis lw 2"],
           [legend,"a = 0.5","a = 1","a = 1.5"])$
```

which produces

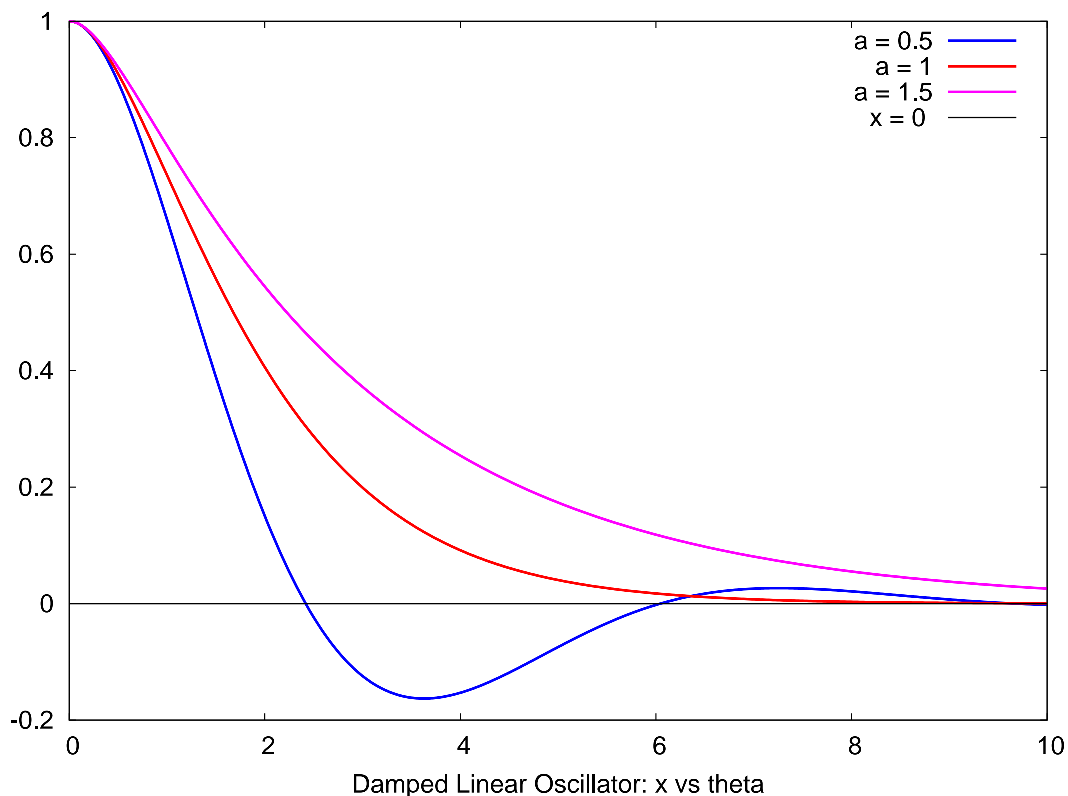

Figure 15: Damped Linear Oscillator

and illustrates why engineers seek the critical damping case, which brings the system to $\mathbf{x} = \mathbf{0}$ most rapidly.

Now for a phase space plot with **dx/dth** versus **x**, drawn for the underdamped case:

```
(%i4) v1 : diff(x[1],th)$
(%i5) fpprintprec:8$
(%i6) [x5,v5] : [x[1],v1],th=5,numer;
(%o6)                        [- 0.0745906, 0.0879424]
(%i7) plot2d ( [ [parametric, x[1], v1, [th,0,10],[nticks,80]],
              [discrete,[[1,0]]], [discrete,[ [x5,v5] ] ] ],
          [x, -0.4, 1.2],[y,-0.8,0.2], [style,[lines,3,7],
                    [points,3,2,1],[points,3,6,1] ],
           [ylabel," "],[xlabel,"th = 0, x = 1, v = 0"],
           [legend," v vs x "," th = 0 "," th = 5 "])$
```

which shows

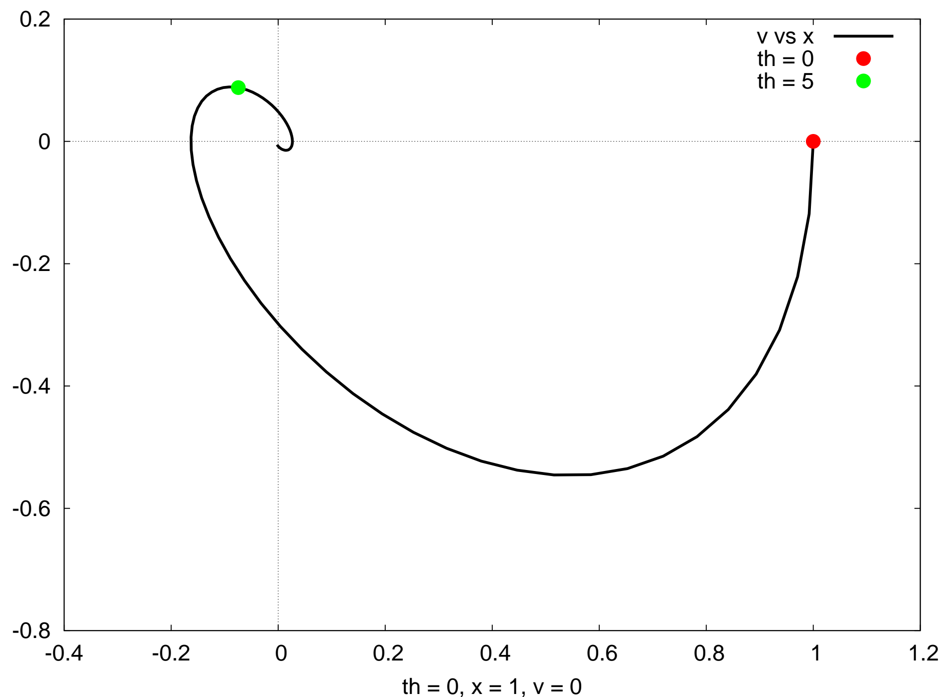

Figure 16: Underdamped Phase Space Plot

Using plotdf for the Damped Linear Oscillator

Let's use **plotdf** to show the phase space plot of our underdamped linear oscillator, using the syntax

```
plotdf ( [dudt, dvdt],[u,v], options...)
```

which requires that we convert our single second order ODE to an equivalent pair of first order ODE's. If we let $\mathbf{d\,x/d\,\theta = v}$, assume the dimensionless damping parameter $\mathbf{a = 1/2}$, we then have $\mathbf{d\,v/d\,\theta = -v - x}$, and we use the **plotdf** syntax

```
plotdf ( [dxdth, dvdth], [x, v], options... ).
```

One has to experiment with the number of steps, the step size, and the horizontal and vertical views. The $\mathbf{v(\theta)}$ values determine the vertical position and the $\mathbf{x(boldsymbol\theta)}$ values determine the horizontal position of a point on the phase space plot curve. The symbols used for the horizontal and vertical ranges should correspond to the symbols used in the second argument (here **[x,v]**). Since we want to get a phase space plot which agrees with our work above, we require the trajectory begin at $\boldsymbol{\theta = 0}, \quad \mathbf{x = 1}, \quad \mathbf{v = 0}$, and we integrate forward in dimensionless time $\boldsymbol{\theta}$.

```
(%i8) plotdf([v,-v-x],[x,v],[trajectory_at,1,0],
           [direction,forward],[x,-0.4,1.2],[v,-0.6,0.2],
            [nsteps,400],[tstep,0.01])$
```

This will bring up the phase space plot **v** vs. **x**, and you can thicken the red curve by clicking the **Config** button (which brings up the **Plot Setup** panel), increasing the **linewidth** to 3, and then clicking **ok**. To actually see the thicker line, you must then click on the **Replot** button. This plot is

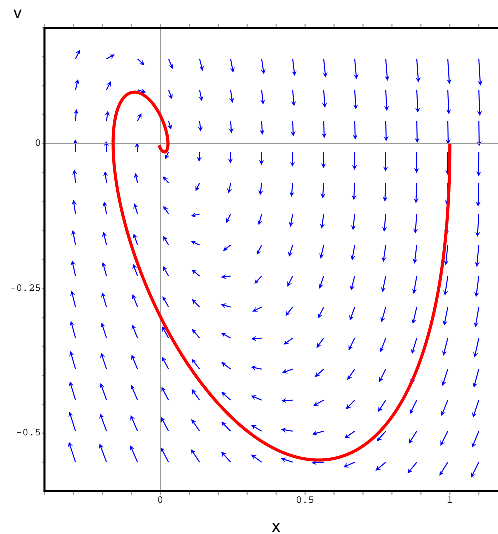

Figure 17: Underdamped Phase Space Plot Using plotdf

To see the separate curves $\mathbf{v}(\theta)$ and $\mathbf{x}(\theta)$, you can click on the **Plot Versus t** button. (The symbol **t** is simply a placeholder for the independent variable, which in our case is θ.) Again, you can change the linewidth and colors (we changed green to red) via the **Config** and **Replot** button process, which yields

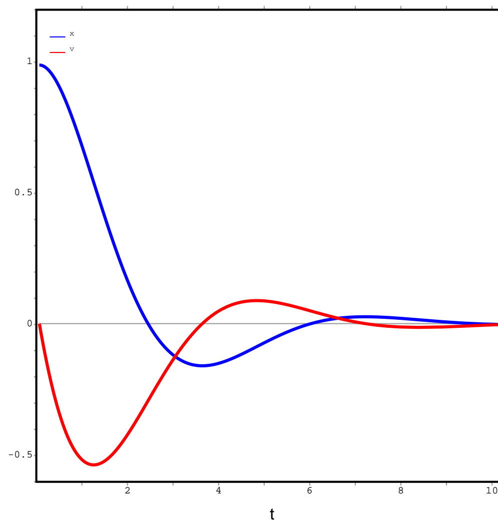

Figure 18: $\mathbf{x}(\theta)$ and $\mathbf{v}(\theta)$ Using plotdf

3.4.5 Ex. 5: Underdamped Linear Oscillator with Sinusoidal Driving Force

We extend our previous oscillator example by adding a sinusoidal driving force. The equation of motion is now

$$m\frac{d^2 x}{d t^2} + b\frac{d x}{d t} + k x = A\cos(\omega t) \tag{3.16}$$

We again divide by the mass m and let

$$\omega_0 = \left(\frac{k}{m}\right)^{1/2}. \tag{3.17}$$

As before, we define

$$\gamma = \frac{b}{2 m}. \tag{3.18}$$

Finally, let $B = A/m$. The equation of motion becomes

$$\frac{d^2 x}{d t^2} + 2\gamma\frac{d x}{d t} + \omega_0^2 x = B\cos(\omega t) \tag{3.19}$$

There are now three natural time scales

$$t1 = \frac{1}{\omega_0}, \quad t2 = \frac{1}{\gamma}, \quad t3 = \frac{1}{\omega} \tag{3.20}$$

and we can introduce a dimensionless time $\theta = \omega_0 t$, the dimensionless positive damping constant $a = \gamma/\omega_0$, the dimensionless oscillator displacement $y = x/B$, and the dimensionless driving angular frequency $q = \omega/\omega_0$ to get

$$\frac{d^2 y}{d\theta^2} + 2 a\frac{d y}{d\theta} + y = \cos(q\,\theta) \tag{3.21}$$

The "underdamped" case corresponds to $\gamma < \omega_0$, or $a < 1$, and we specialize to the case $a = 1/2$.

```
(%i1) de : 'diff(y,th,2) + 'diff(y,th) + y - cos(q*th);
                                 2
                                d y     dy
(%o1)                           ---- + --- + y - cos(q th)
                                   2   dth
                                dth
(%i2) gsoln : ode2(de,y,th);
                                    2
           q sin(q th) + (1 - q ) cos(q th)
(%o2) y = --------------------------------
                     4     2
                    q   - q   + 1
                                 - th/2          sqrt(3) th            sqrt(3) th
                         + %e        (%k1 sin(----------) + %k2 cos(----------))
                                                    2                     2
(%i3) psoln : ic2(gsoln,th=0,y=1,diff(y,th)=0);
                                    2
           q sin(q th) + (1 - q ) cos(q th)
(%o3) y = --------------------------------
                     4     2
                    q   - q   + 1
                              4      2       sqrt(3) th         4      sqrt(3) th
                          (q   - 2 q ) sin(----------)       q   cos(----------)
                 - th/2                         2                       2
            + %e       (-------------------------------- + ------------------)
                                4               2                4     2
                        sqrt(3) q   - sqrt(3) q   + sqrt(3)     q   - q   + 1
```

We now specialize to a high (dimensionless) driving angular frequency case, $\mathbf{q} = \mathbf{4}$, which means that we are assuming that the actual driving angular frequency is four times as large as the natural angular frequency of this oscillator.

```
(%i4) ys : subst(q=4,rhs(psoln));
                          sqrt(3) th              sqrt(3) th
                    224 sin(----------)    256 cos(----------)
        - th/2                  2                      2
(%o4) %e          (-------------------- + --------------------)
                       241 sqrt(3)                241
                                                      4 sin(4 th) - 15 cos(4 th)
                                                    + --------------------------
                                                                241
(%i5) vs : diff(ys,th)$
```

We now plot both the dimensionless oscillator amplitude and the dimensionless oscillator velocity on the same plot.

```
(%i6) plot2d([ys,vs],[th,0,12],
        [nticks,100],
        [style,[lines,5]],
        [legend," Y "," V "],
        [xlabel," dimensionless Y and V vs. theta"])$
```

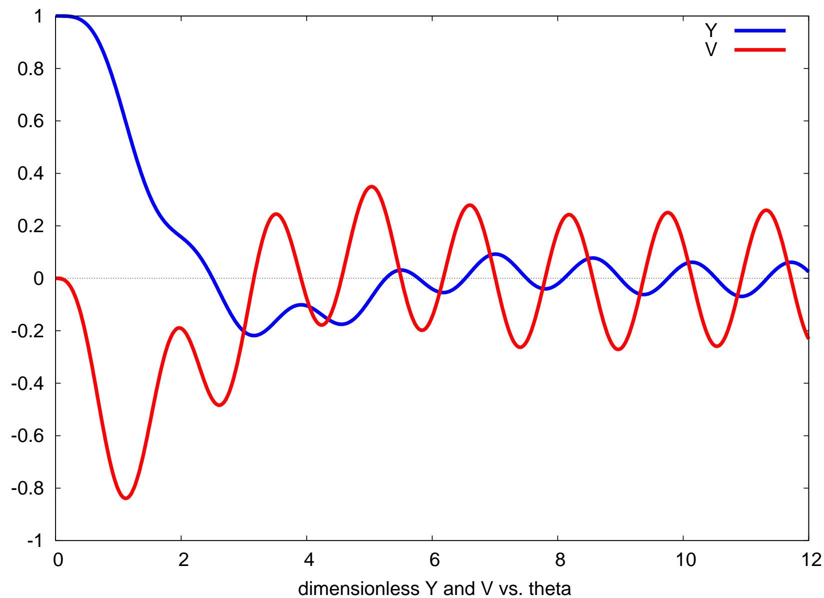

Figure 19: Dimensionless Y and V versus Dimensionless Time θ

We see that the driving force soon dominates the motion of the underdamped linear oscillator, which is forced to oscillate at the driving frequency. This dominance evidently has nothing to do with the actual strength **A newtons** of the peak driving force, since we are solving for a dimensionless oscillator amplitude, and we get the same qualitative curve no matter what the size of **A** is.

We next make a phase space plot for the early "capture" part of the motion of this system. (Note that **plotdf** cannot numerically integrate this differential equation because of the explicit appearance of the dependent variable.)

```
(%i7) plot2d([parametric,ys,vs,[th,0,8]],
          [style,[lines,5]],[nticks,100],
           [xlabel," V (vert) vs. Y (hor) "])$
```

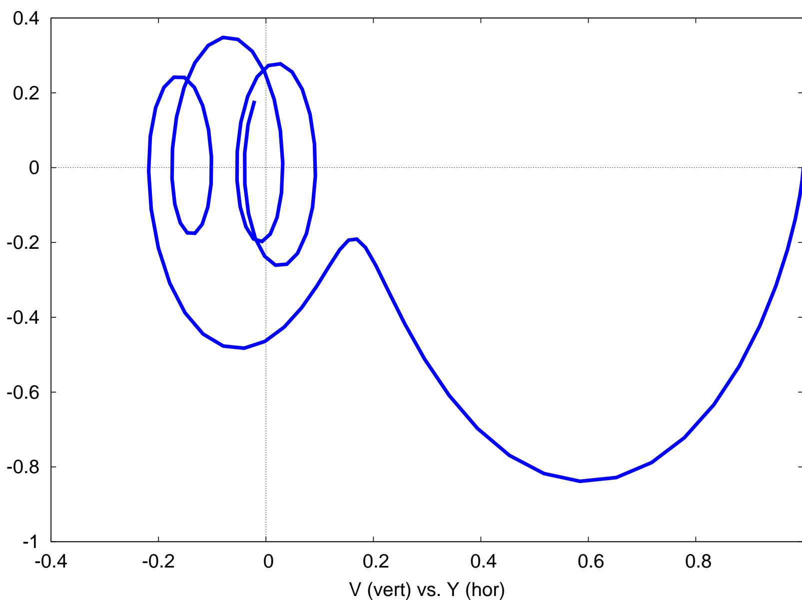

Figure 20: Dimensionless V Versus Dimensionless Y: Early History

We see the phase space plot being driven to regular oscillations about **y = 0** and **v = 0**.

3.4.6 Ex. 6: Regular and Chaotic Motion of a Driven Damped Planar Pendulum

The motion is pure rotation in a fixed plane (one degree of freedom), and if the pendulum is a simple pendulum with all the mass **m** concentrated at the end of a weightless support of length **L**, then the moment of inertia about the support point is $\mathbf{I} = \mathbf{m\,L^2}$, and the angular acceleration is $\boldsymbol{\alpha}$, and rotational dynamics implies the equation of motion

$$\mathbf{I}\,\boldsymbol{\alpha} = \mathbf{m\,L^2}\,\frac{\mathbf{d^2}\boldsymbol{\theta}}{\mathbf{d\,t^2}} = \boldsymbol{\tau}_\mathbf{z} = -\mathbf{m\,g\,L}\sin\boldsymbol{\theta} - \mathbf{c}\,\frac{\mathbf{d}\,\boldsymbol{\theta}}{\mathbf{d\,t}} + \mathbf{A}\cos(\boldsymbol{\omega}_\mathbf{d}\,\mathbf{t}) \tag{3.22}$$

We introduce a dimensionless time $\boldsymbol{\tau} = \boldsymbol{\omega}_\mathbf{0}\,\mathbf{t}$ and a dimensionless driving angular frequency $\boldsymbol{\omega} = \boldsymbol{\omega}_\mathbf{d}/\boldsymbol{\omega}_\mathbf{0}$, where $\boldsymbol{\omega}_\mathbf{0}^\mathbf{2} = \mathbf{g}/\mathbf{L}$, to get the equation of motion

$$\frac{\mathbf{d^2}\boldsymbol{\theta}}{\mathbf{d}\,\boldsymbol{\tau}^\mathbf{2}} = -\sin\boldsymbol{\theta} - \mathbf{a}\,\frac{\mathbf{d}\,\boldsymbol{\theta}}{\mathbf{d}\,\boldsymbol{\tau}} + \mathbf{b}\cos(\boldsymbol{\omega}\,\boldsymbol{\tau}) \tag{3.23}$$

To simplify the notation for our exploration of this differential equation, we make the replacements $\boldsymbol{\theta} \to \mathbf{u}$, $\boldsymbol{\tau} \to \mathbf{t}$, and $\boldsymbol{\omega} \to \mathbf{w}$ (parameters **a**, **b**, and **w** are dimensionless) to work with the differential equation:

$$\frac{\mathbf{d^2\,u}}{\mathbf{d\,t^2}} = -\sin\mathbf{u} - \mathbf{a}\,\frac{\mathbf{d\,u}}{\mathbf{d\,t}} + \mathbf{b}\cos(\mathbf{w\,t}) \tag{3.24}$$

where now both **t** and **u** are dimensionless, with the measure of **u** being radians, and the physical values of the pendulum angle being limited to the range $-\boldsymbol{\pi} \le \mathbf{u} \le \boldsymbol{\pi}$, both extremes being the "flip-over-point" at the top of the motion.

We will use both **plotdf** and **rk** to explore this system, with

$$\frac{\mathbf{d\,u}}{\mathbf{d\,t}} = \mathbf{v}, \quad \frac{\mathbf{d\,v}}{\mathbf{d\,t}} = -\sin\mathbf{u} - \mathbf{a\,v} + \mathbf{b}\cos(\mathbf{w\,t}) \tag{3.25}$$

3.4.7 Free Oscillation Case

Using **plotdf**, the phase space plot for NO friction and NO driving torque is

```
(%i1) plotdf([v,-sin(u)],[u,v],[trajectory_at,float(2*%pi/3),0],
             [direction,forward],[u,-2.5,2.5],[v,-2.5,2.5],
              [tstep, 0.01],[nsteps,600])$
```

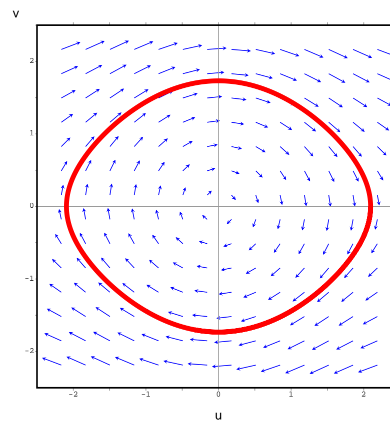

Figure 21: No Friction, No Driving Torque: V Versus Angle U

and now we use the **Plot Versus t** button of **plotdf** to show the angle **u radians** and the dimensionless rate of change of angle **v**

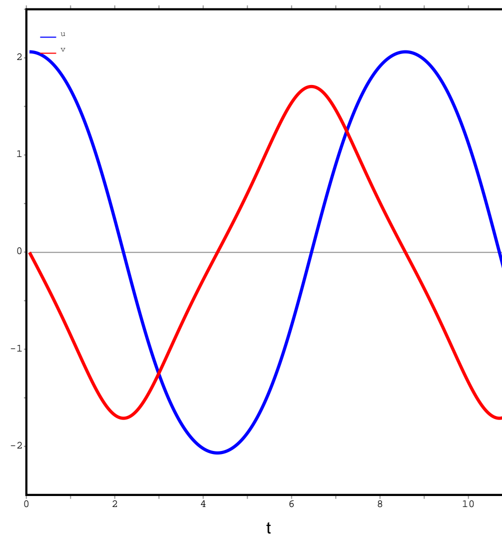

Figure 22: No Friction, No Driving Torque: Angle U [blue] and V [red]

3.4.8 Damped Oscillation Case

We now include some damping with $\mathbf{a} = \mathbf{1/2}$.

```
(%i2)  plotdf([v,-sin(u)-0.5*v],[u,v],[trajectory_at,float(2*%pi/3),0],
          [direction,forward],[u,-1,2.5],[v,-1.5,1],
           [tstep, 0.01],[nsteps,450])$
```

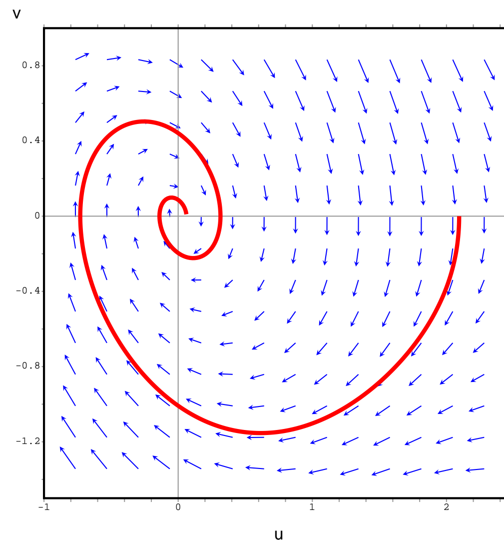

Figure 23: With Friction, but No Driving Force: V Versus Angle U

and now we use the **Plot Versus t** button of **plotdf** to show the angle **u radians** and the dimensionless rate of change of angle **y** for the friction present case.

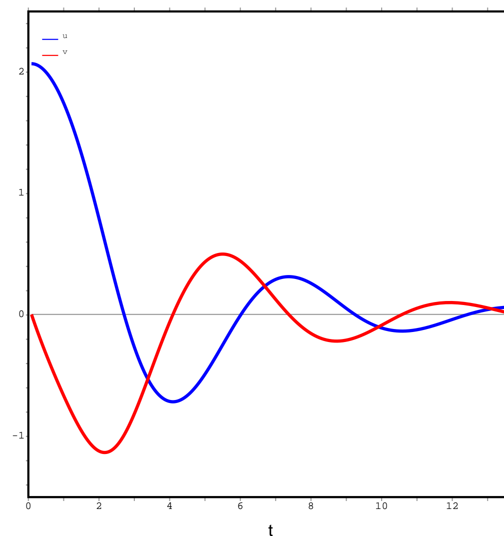

Figure 24: With Friction, but No Driving Force: Angle U [blue] and V [red]

3.4.9 Including a Sinusoidal Driving Torque

We now use the Runge-Kutta function **rk** to integrate the differential equation set forward in time for **ncycles**, which is the same as setting the final dimensionless **tmax** equal to **ncycles*2*%pi/w**, or **ncycles*T**, where we can call **T** the dimensionless period defined by the dimensionless angular frequency **w**. The physical meaning of **T** is the ratio of the period of the driving torque to the period of unforced and undamped small oscillations of the free simple pendulum.

For simplicity of exposition, we will call **t** the "time" and **T** the "period". We again use our homemade function **fll** described in the preface.

One cycle (period) of time is divided into **nsteps** subdivisions, so **dt = T/nsteps**.

For both the regular and chaotic parameter cases, we have used the same parameters as used in **Mathematica in Theoretical Physics**, by Gerd Baumann, Springer/Telos, 1996, pages 46 - 53.

3.4.10 Regular Motion Parameters Case

We find regular motion of this driven system with **a = 0.2, b = 0.52, and w = 0.694**, and with **u0 = 0.8 rad**, and **v0 = 0.8 rad/unit-time**.

```
(%i1) fpprintprec:8$
(%i2) (nsteps : 31, ncycles : 30, a : 0.2, b : 0.52, w : 0.694)$
(%i3) [dudt : v, dvdt : -sin(u) - a*v + b*cos(w*t),
          T : float(2*%pi/w ) ];
(%o3)        [v, - 0.2 v - sin(u) + 0.52 cos(0.694 t), 9.0535811]
(%i4) [dt : T/nsteps, tmax : ncycles*T ];
(%o4)                          [0.292051, 271.60743]
(%i5) tuvL : rk ([dudt,dvdt],[u,v],[0.8,0.8],[t,0,tmax, dt])$
(%i6) %, fll;
(%o6)      [[0, 0.8, 0.8], [271.60743, - 55.167003, 1.1281164], 931]
(%i7) 930*dt;
(%o7)                                271.60743
```

Plot of u(t) and v(t)

Plot of u(t) and v(t) against t

```
(%i8) tuL : makelist ([tuvL[i][1],tuvL[i][2]],i,1,length(tuvL))$
(%i9) %, fll;
(%o9)              [[0, 0.8], [271.60743, - 55.167003], 931]
(%i10) tvL : makelist ([tuvL[i][1],tuvL[i][3]],i,1,length(tuvL))$
(%i11) %, fll;
(%o11)              [[0, 0.8], [271.60743, 1.1281164], 931]
(%i12) plot2d([ [discrete,tuL], [discrete,tvL]],[x,0,280],
          [style,[lines,3]],[xlabel,"t"],
           [legend, "u", "v"],
             [gnuplot_preamble,"set key bottom left;"])$
```

which produces

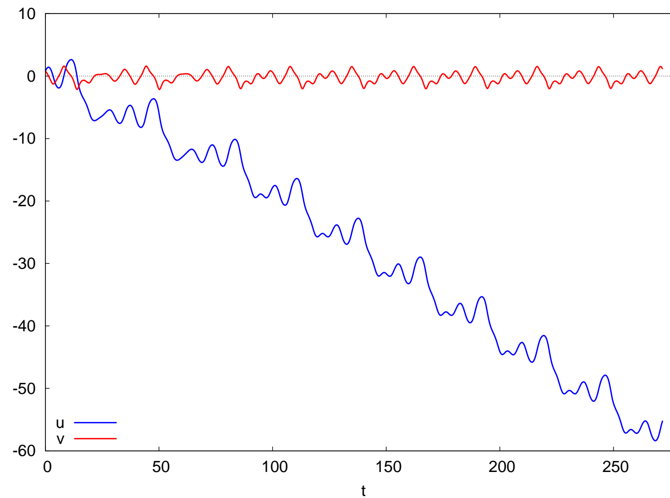

Figure 25: Angle u(t) and v(t)

The above plot shows nine flips of the pendulum at the top:
the first passage over the top at **u = -3 pi/2 = -4.7 rad**,
the second passage over the top at **u = -7 pi/2 = -11 rad**,
and so on.

Phase Space Plot

We next construct a phase space plot.

```
(%i13) uvL : makelist ([tuvL[i][2],tuvL[i][3]],i,1,length(tuvL))$
(%i14) %, fll;
(%o14)            [[0.8, 0.8], [- 55.167003, 1.1281164], 931]
(%i15) plot2d ( [discrete,uvL],[x,-60,5],[y,-5,5],
                 [style,[lines,3]],
             [ylabel," "],[xlabel," v vs u "] )$
```

which produces (note that we include the early points which are more heavily influenced by the initial conditions):

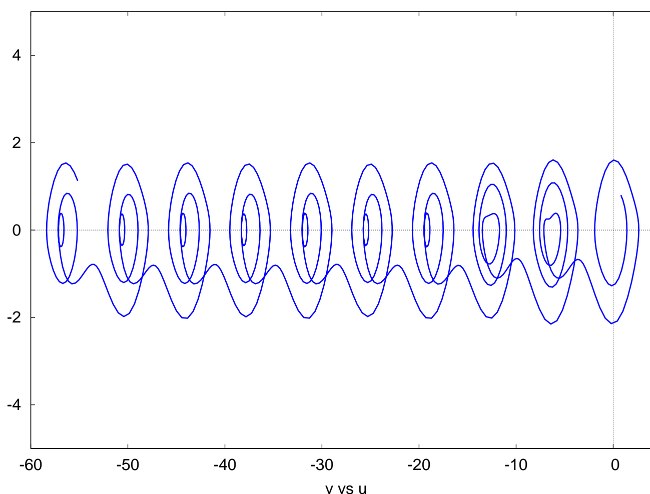

Figure 26: Non-Reduced Phase Space Plot

Reduced Phase Space Plot

Let's define a Maxima function **reduce** which brings **u** back to the interval **(-pi, pi)** and then make a reduced phase space plot. Since this is a strictly numerical task, we can simplify Maxima's efforts by defining a floating point number **pi** once and for all, and simply work with that definition. You can see the origin of our definition of **reduce** in the manual's entry on Maxima's modulus function **mod**.

```
(%i16) pi : float(%pi);
(%o16)                          3.1415927
(%i17) reduce(yy) := pi - mod (pi - yy,2*pi)$
(%i18) float( [-7*%pi/2,-3*%pi/2 ,3*%pi/2, 7*%pi/2] );
(%o18)          [- 10.995574, - 4.712389, 4.712389, 10.995574]
(%i19) map('reduce, % );
(%o19)         [1.5707963, 1.5707963, - 1.5707963, - 1.5707963]
(%i20) uvL_red : makelist ( [ reduce( tuvL[i][2]),
                  tuvL[i][3]],i,1,length(tuvL))$
(%i21) %, fll;
(%o21)            [[0.8, 0.8], [1.3816647, 1.1281164], 931]
```

To make a reduced phase space plot with our reduced regular motion points, we will only use the last two thirds of the pairs **(u,v)**. This will then show the part of the motion which has been "captured" by the driving torque and shows little influence of the initial conditions.

We use the Maxima function **rest (list, n)** which returns **list** with its first **n** elements removed if **n** is positive. Thus we use **rest (list, num/3)** to get the last two thirds.

```
(%i22) uvL_regular : rest (uvL_red, round(length (uvL_red)/3) )$
(%i23) %, fll;
(%o23)      [[0.787059, - 1.2368529], [1.3816647, 1.1281164], 621]
(%i24) plot2d ( [discrete,uvL_regular],[x,-3.2,3.2],[y,-3.2,3.2],
            [style,[lines,2]],
        [ylabel," "],[xlabel,"reduced phase space v vs u "] )$
```

which produces

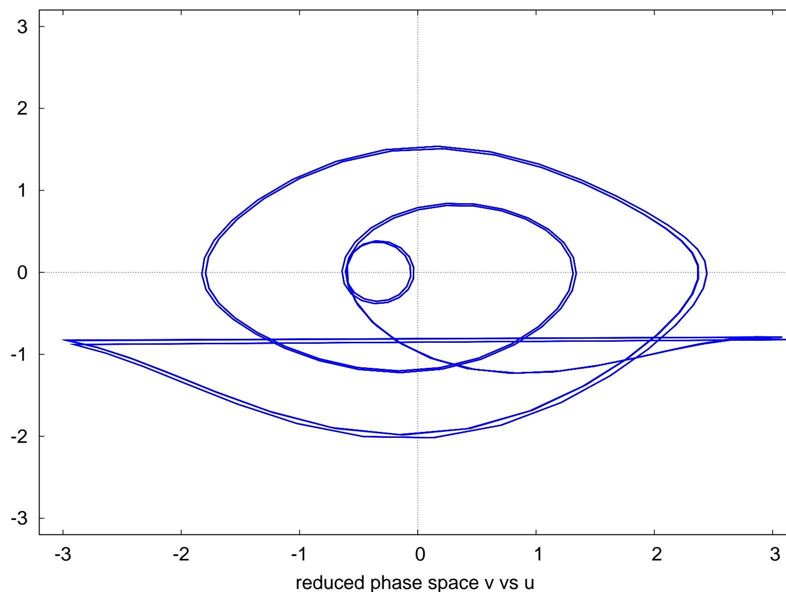

Figure 27: Reduced Phase Space Plot of Regular Motion Points

Poincare Plot

We next construct a Poincare plot of the regular (reduced) phase space points by using a "stroboscopic view" of this phase space, displaying only phase space points which correspond to times separated by the driving period **T**. We select `(u,v)` pairs which correspond to intervals of time `n*T`, where `n = 10, 11, ..., 30` which will give us `21` phase space points for our plot (this is roughly the same as taking the last two thirds of the points).

The time `t = 30*T` corresponds to `t = 30*31*dt` = `930*dt` which is the time associated with element `931`, the last element) `of uvL_red`. The value of `j` used to select the last Poincare point is the solution of the equation `1 + 10*nsteps + j*nsteps = 1 + ncycles*nsteps`, which for this case is equivalent to
`311 + j*31 = 931`.

```
(%i25) solve(311 + j*31 = 931);
(%o25)                              [j = 20]
(%i26) pL : makelist (1+10*nsteps + j*nsteps, j, 0, 20);
(%o26) [311, 342, 373, 404, 435, 466, 497, 528, 559, 590, 621, 652, 683, 714,
                                          745, 776, 807, 838, 869, 900, 931]
(%i27) length(pL);
(%o27)                                 21
(%i28) poincareL : makelist (uvL_red[i], i, pL)$
(%i29) %,fll;
(%o29)       [[0.787059, - 1.2368529], [1.3816647, 1.1281164], 21]
(%i30) plot2d ( [discrete,poincareL],[x,-0.5,2],[y,-1.5,1.5],
                    [style,[points,1,1,1 ]],
              [ylabel," "],[xlabel," Poincare Section v vs u "] )$
```

which produces the plot

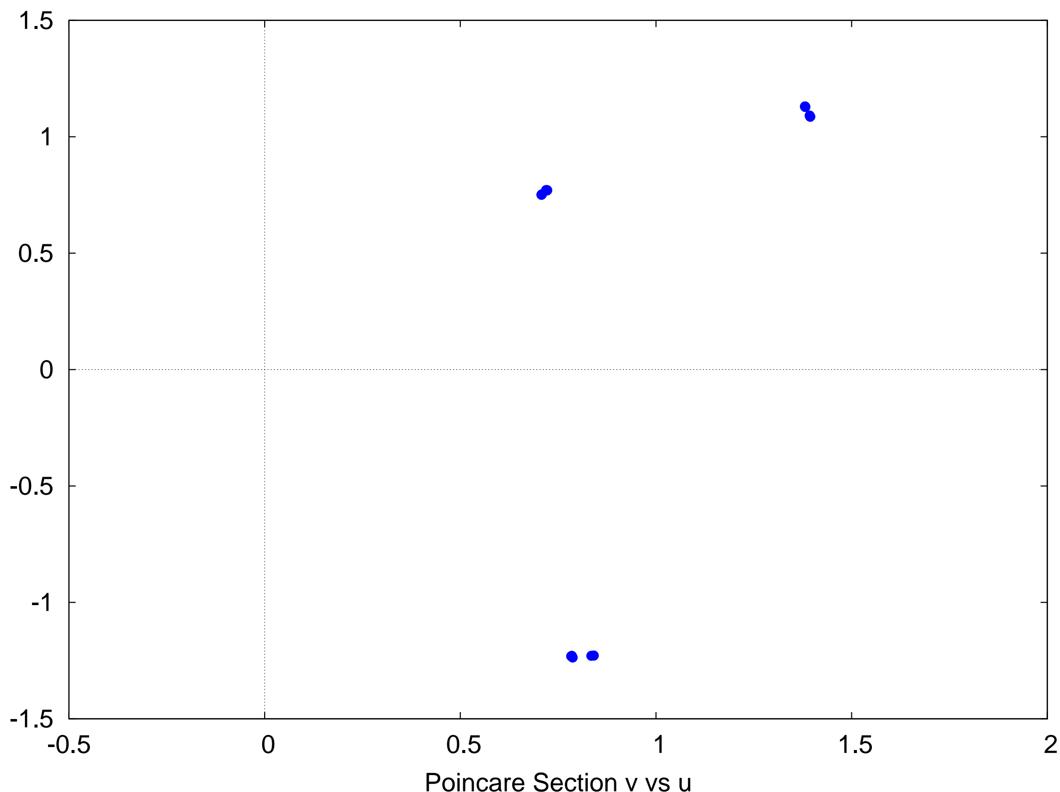

Figure 28: Reduced Phase Space Plot of Regular Motion Points

For this regular motion parameters case, the Poincare plot shows the phase space point coming back to one of three general locations in phase space at times separated by the period **T**.

3.4.11 Chaotic Motion Parameters Case.

To exhibit an example of chaotic motion for this system, we use the same initial conditions for **u** and **v**, but use the parameter set **a = 1/2, b = 1.15, w = 2/3**.

```
(%i1) fpprintprec:8$
(%i2) (nsteps : 31, ncycles : 240, a : 1/2, b : 1.15, w : 2/3)$
(%i3) [dudt : v, dvdt : -sin(u) - a*v + b*cos(w*t),
           T : float(2*%pi/w ) ];
                          v                      2 t
(%o3)               [v, - - - sin(u) + 1.15 cos(---), 9.424778]
                          2                       3
(%i4) [dt : T/nsteps, tmax : ncycles*T ];
(%o4)                         [0.304025, 2261.9467]
(%i5) tuvL : rk ([dudt,dvdt],[u,v],[0.8,0.8],[t,0,tmax, dt])$
(%i6) %, fll;
(%o6)       [[0, 0.8, 0.8], [2261.9467, 26.374502, 0.937008], 7441]
(%i7) dt*( last(%) - 1 );
(%o7)                               2261.9467
(%i8) tuL : makelist ([tuvL[i][1],tuvL[i][2]],i,1,length(tuvL))$
(%i9) %, fll;
(%o9)               [[0, 0.8], [2261.9467, 26.374502], 7441]
(%i10) tvL : makelist ([tuvL[i][1],tuvL[i][3]],i,1,length(tuvL))$
(%i11) %, fll;
(%o11)               [[0, 0.8], [2261.9467, 0.937008], 7441]
(%i12) plot2d([ [discrete,tuL], [discrete,tvL]],[x,0,2000],
                [y,-15,30],
           [style,[lines,2]],[xlabel,"t"], [ylabel, " "],
            [legend, "u","v" ] ,[gnuplot_preamble,"set key bottom;"])$
```

which produces a plot of **u(t)** and **v(t)** over **0 <= t <= 2000**:

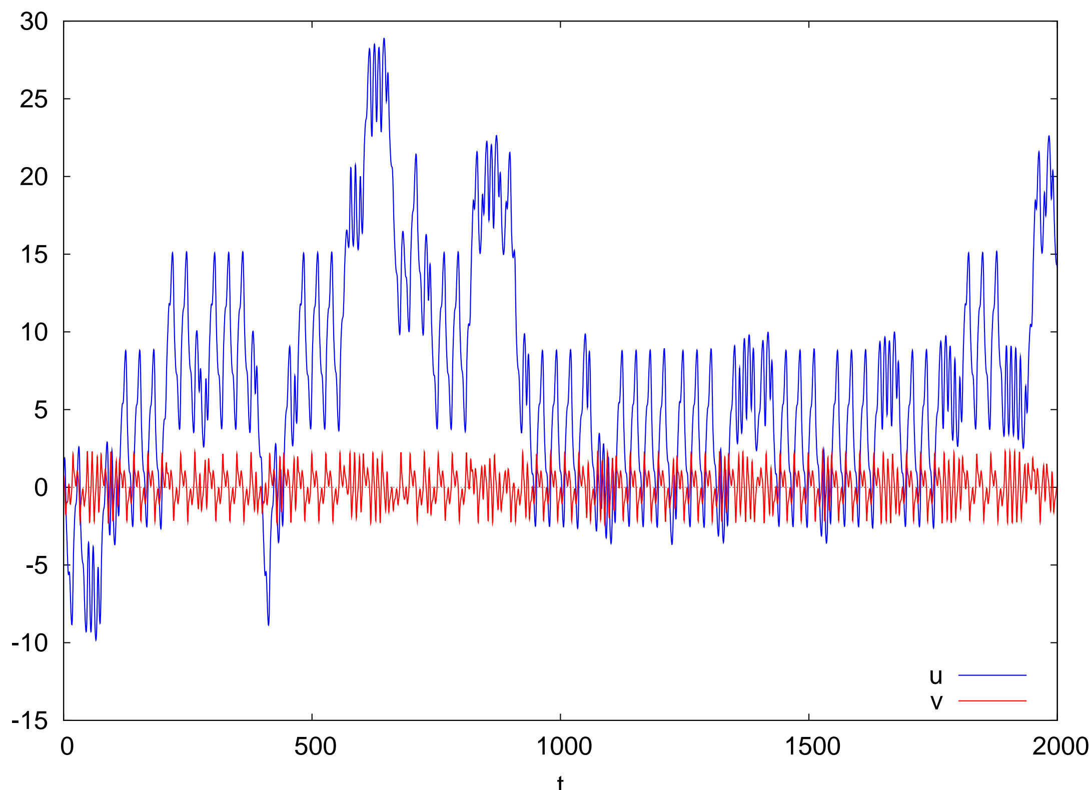

Figure 29: Angle u(t), and v(t) for Chaotic Parameters Choice

Phase Space Plot

We next construct a non-reduced phase space plot, but show only the first `2000` reduced phase space points.

```
(%i13) uvL : makelist ([tuvL[i][2],tuvL[i][3]],i,1,length(tuvL))$
(%i14) %, fll;
(%o14)              [[0.8, 0.8], [26.374502, 0.937008], 7441]
(%i15) uvL_first : rest(uvL, -5441)$
(%i16) %, fll;
(%o16)              [[0.8, 0.8], [23.492001, 0.299988], 2000]
(%i17) plot2d ( [discrete,uvL_first],[x,-12,30],[y,-3,3],
                  [style,[points,1,1,1]],
            [ylabel," "],[xlabel," v vs u "])$
```

which produces

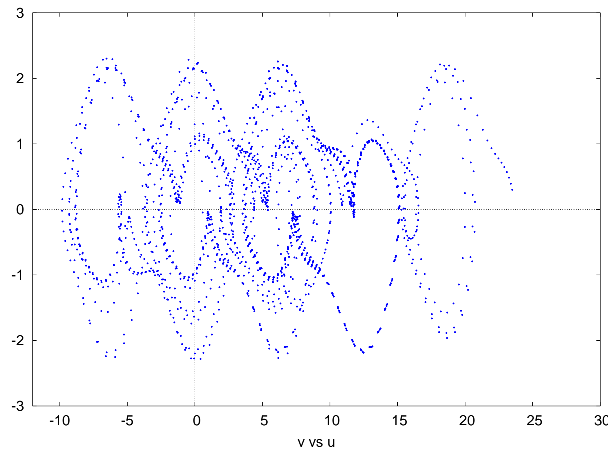

Figure 30: non-reduced phase space plot using first 2000 points

If we use the **discrete** default **style** option `lines` instead of `points`,

```
(%i18) plot2d ( [discrete,uvL_first],[x,-12,30],[y,-3,3],
            [ylabel," "],[xlabel," v vs u "])$
```

we get the non-reduced phase space plot drawn with lines between the points:

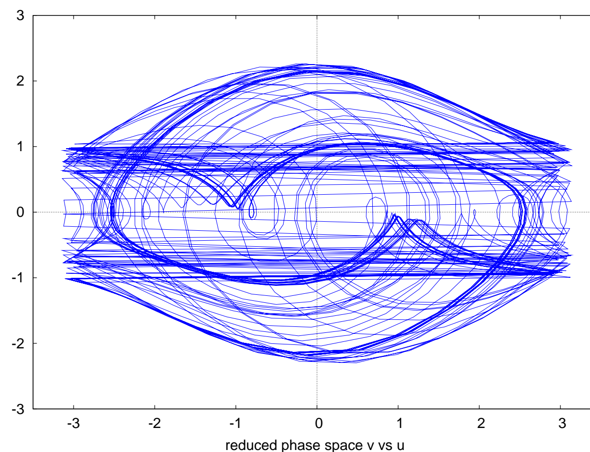

Figure 31: non-reduced phase space plot using first 2000 points

Reduced Phase Space Plot

We now construct the reduced phase space points as in the regular motion case and then omit the first **400**.

```
(%i19) pi : float(%pi);
(%o19)                             3.1415927
(%i20) reduce(yy) := pi - mod (pi - yy,2*pi)$
(%i21) uvL_red : makelist ( [ reduce( first( uvL[i] )),
                   second( uvL[i] ) ],i,1,length(tuvL))$
(%i22) %, fll;
(%o22)             [[0.8, 0.8], [1.2417605, 0.937008], 7441]
(%i23) uvL_cut : rest(uvL_red, 400)$
(%i24) %, fll;
(%o24)        [[0.25464, 1.0166641], [1.2417605, 0.937008], 7041]
```

We have discarded the first **400** reduced phase space points in defining **uvL_cut**. If we now only plot the first **1000** of the points retained in **uvL_cut**:

```
(%i25) uvL_first : rest (uvL_cut, -6041)$
(%i26) %, fll;
(%o26)        [[0.25464, 1.0166641], [2.2678603, 0.608686], 1000]
(%i27) plot2d ( [discrete,uvL_first],[x,-3.5,3.5],[y,-3,3],
                [style,[points,1,1,1]],
             [ylabel," "],[xlabel,"reduced phase space v vs u "])$
```

we get the plot

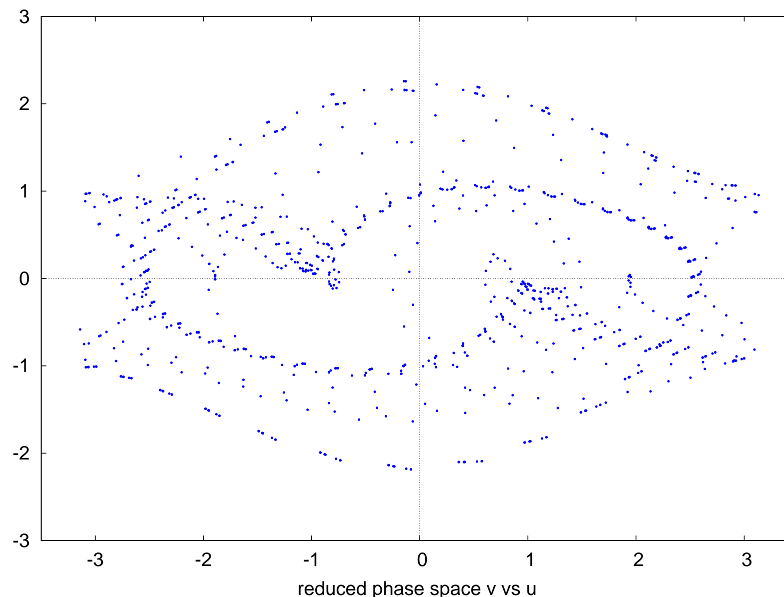

Figure 32: 1000 points reduced phase space plot

and the same set of points drawn with the default **lines** option:

```
(%i28) plot2d ( [discrete,uvL_first],[x,-3.5,3.5],[y,-3,3],
             [ylabel," "],[xlabel,"reduced phase space v vs u "])$
```

produces the plot

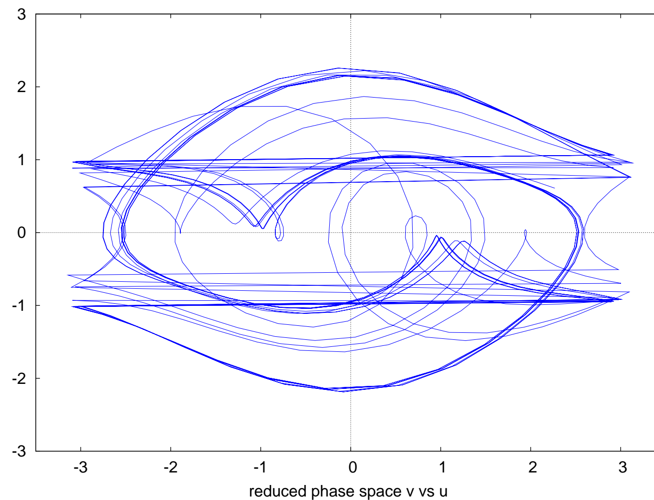

Figure 33: 1000 points reduced phase space plot

3000 point phase space plot

We next draw the same reduced phase space plot, but use the first **3000** points of **uvL_cut**.

```
(%i29) uvL_first : rest (uvL_cut, -4041)$
(%i30) %, fll;
(%o30)      [[0.25464, 1.0166641], [- 2.2822197, - 0.532184], 3000]
(%i31) plot2d ( [discrete,uvL_first],[x,-3.5,3.5],[y,-3,3],
                [style,[points,1,1,1]],
            [ylabel," "],[xlabel,"reduced phase space v vs u "])$
```

which produces

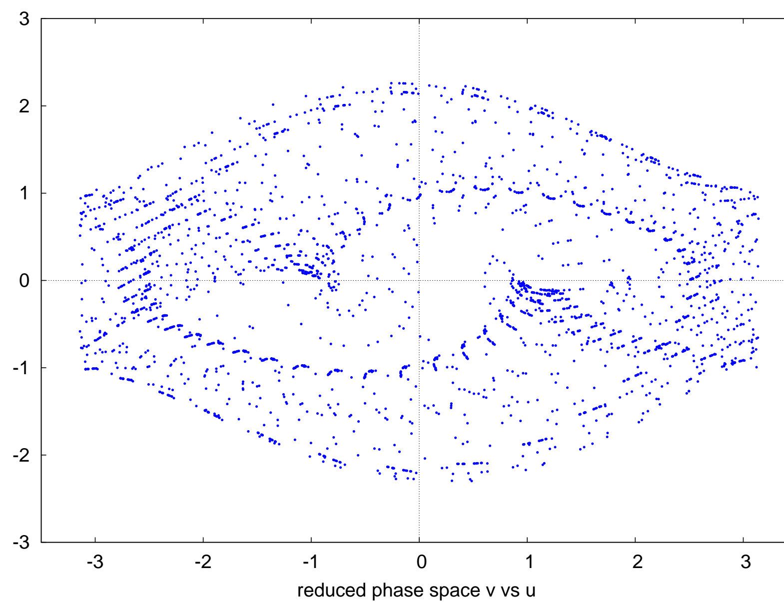

Figure 34: 3000 points reduced phase space plot

and again, the same set of points drawn with the default **lines** option

```
(%i32) plot2d ( [discrete,uvL_first],[x,-3.5,3.5],[y,-3,3],
            [ylabel," "],[xlabel,"reduced phase space v vs u "])$
```

which produces the plot

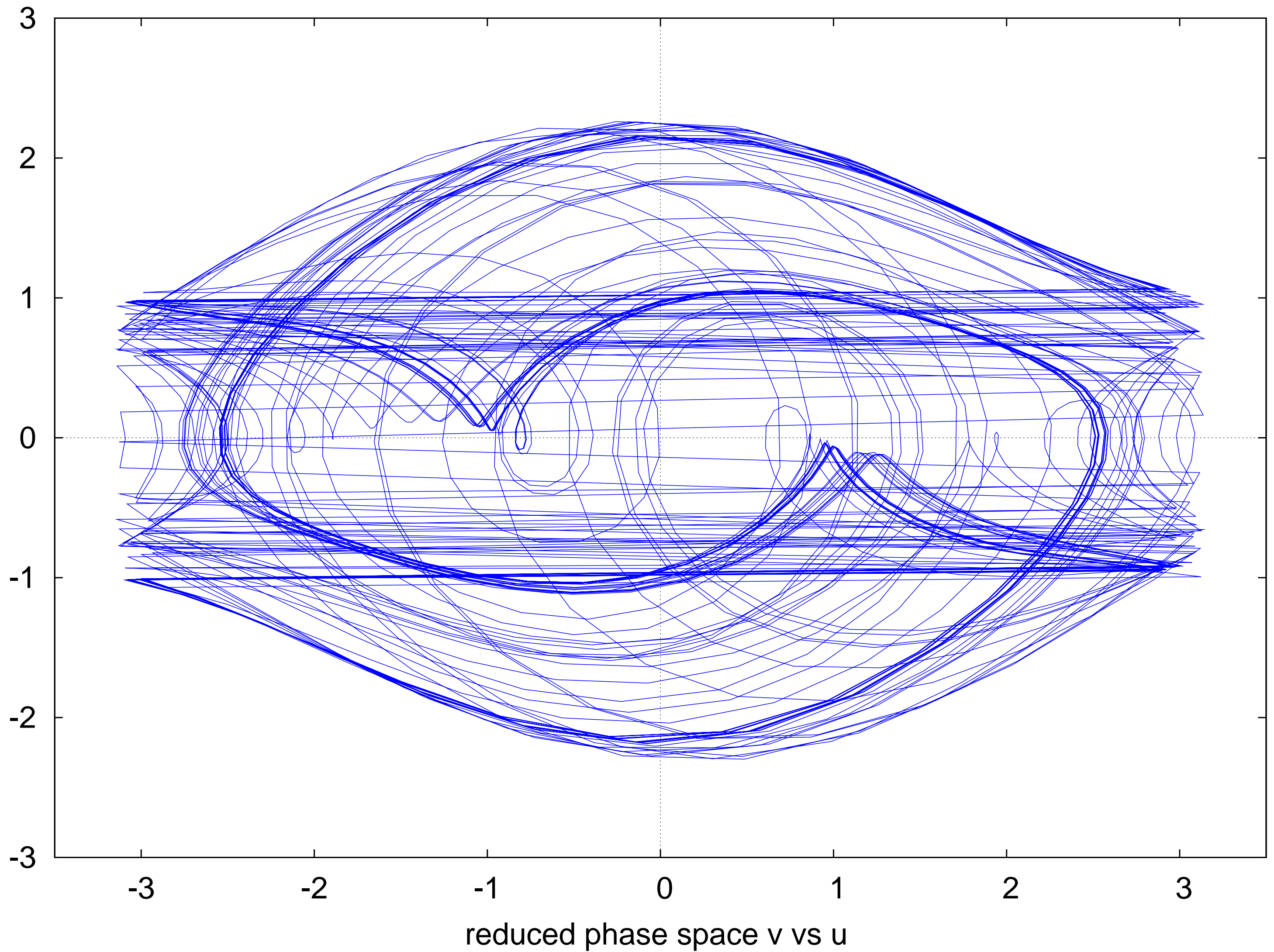

Figure 35: 3000 points reduced phase space plot

Poincare Plot

We now construct the Poincare section plot as before, using all the points available in **uvL_red**.

```
(%i33) pL : makelist(1+10*nsteps + j*nsteps, j, 0, ncycles - 10)$
(%i34) %, fll;
(%o34)                          [311, 7441, 231]
(%i35) poincareL : makelist(uvL_red[i], i, pL)$
(%i36) %, fll;
(%o36)       [[- 2.2070801, 1.3794391], [1.2417605, 0.937008], 231]
(%i37) plot2d ( [discrete,poincareL],[x,-3,3],[y,-4,4],
                    [style,[points,1,1,1]],
             [ylabel," "],[xlabel," Poincare Section v vs u "] )$
```

which produces the plot

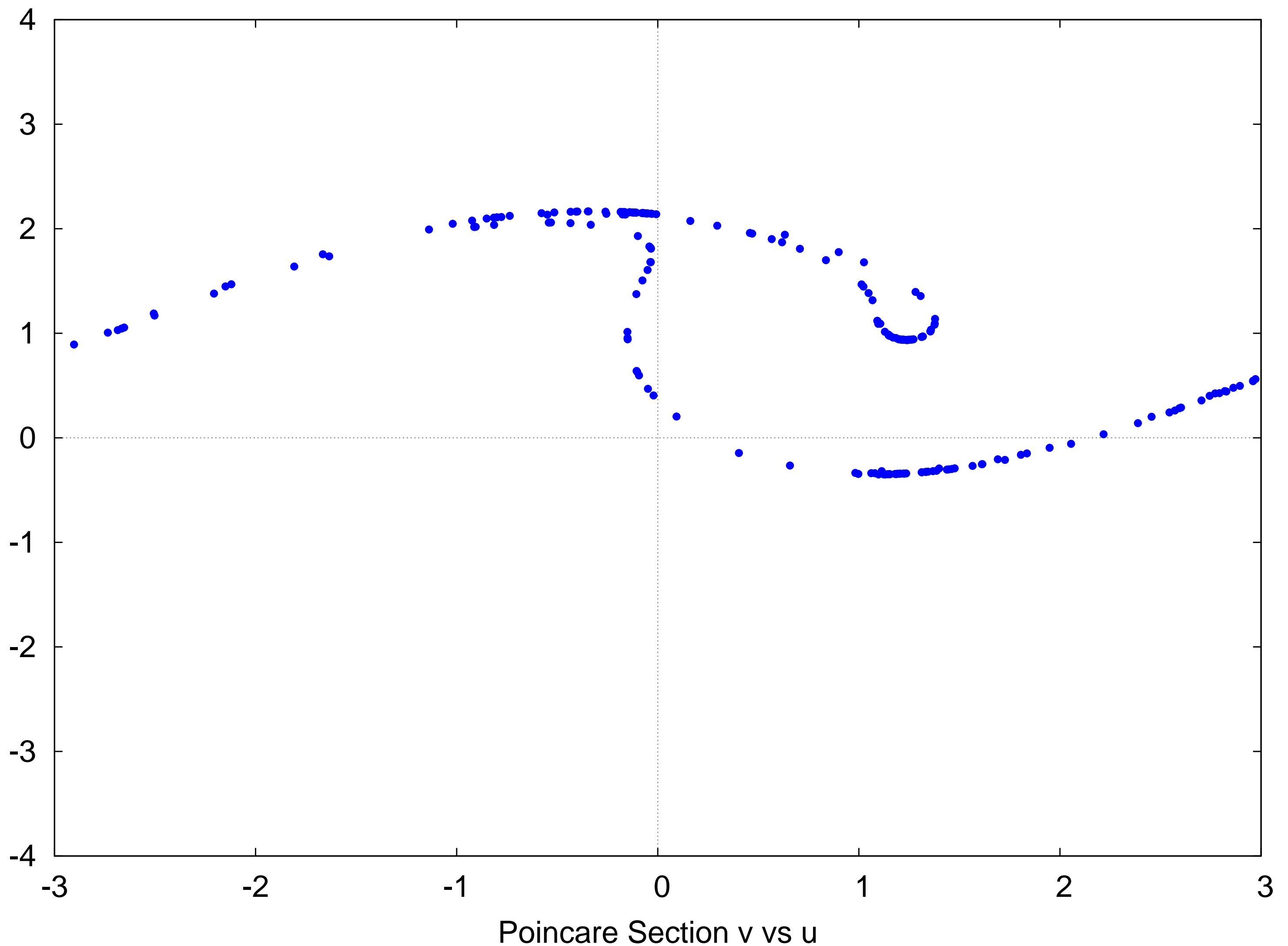

Figure 36: 231 poincare section points

3.5 Using contrib_ode for ODE's

The syntax of **contrib_ode** is the same as **ode**. Let's first solve the same first order ODE example used in the first sections.

```
(%i1) de : 'diff(u,t)- u - exp(-t);
                                du         - t
(%o1)                           -- - u - %e
                                dt
(%i2) gsoln : ode2(de,u,t);
                                        - 2 t
                                      %e         t
(%o2)                       u = (%c - -------) %e
                                         2
(%i3) contrib_ode(de,u,t);
                                   du         - t
(%o3)                   contrib_ode(-- - u - %e    , u, t)
                                   dt
(%i4) load('contrib_ode);
(%o4) C:/PROGRA~1/MAXIMA~3.1/share/maxima/5.18.1/share/contrib/diffequations/c\
ontrib_ode.mac
(%i5) contrib_ode(de,u,t);
                                        - 2 t
                                      %e         t
(%o5)                      [u = (%c - -------) %e ]
                                         2
(%i6) ode_check(de, %[1] );
(%o6)                                   0
```

We see that **contrib_ode**, with the same syntax as **ode**, returns a list (here with one solution, but in general more than one solution) rather than simply one answer.

Moreover, the package includes the Maxima function **ode_check** which can be used to confirm the general solution.

Here is a comparison for our second order ODE example.

```
(%i7) de : 'diff(u,t,2) - 4*u;
                                   2
                                  d u
(%o7)                             --- - 4 u
                                    2
                                  dt
(%i8) gsoln : ode2(de,u,t);
                                   2 t         - 2 t
(%o8)                     u = %k1 %e    + %k2 %e
(%i9) contrib_ode(de,u,t);
                                   2 t         - 2 t
(%o9)                    [u = %k1 %e    + %k2 %e     ]
(%i10) ode_check(de, %[1] );
(%o10)                                  0
```

Here is an example of an ODE which **ode2** cannot solve, but **contrib_ode** can solve.

```
(%i11) de : 'diff(u,t,2) + 'diff(u,t) + t*u;
                                2
                               d u   du
(%o11)                         --- + -- + t u
                                 2   dt
                               dt
(%i12) ode2(de,u,t);
(%o12)                                false
```

```
(%i13) gsoln : contrib_ode(de,u,t);
                                      3/2
                     1   (4 t - 1)                            - t/2
(%o13) [u = bessel_y(-, ------------) %k2 sqrt(4 t - 1) %e
                     3       12
                                                   3/2
                                    1   (4 t - 1)                            - t/2
                         + bessel_j(-, ------------) %k1 sqrt(4 t - 1) %e     ]
                                    3       12
(%i14) ode_check(de, %[1] );
(%o14)                                 0
```

This section will probably be augmented in the future with more examples of using **contrib_ode**.

Maxima by Example: Ch.4: Solving Equations *

Edwin L. Woollett

January 29, 2009

Contents

4 Solving Equations **3**

4.1 One Equation or Expression: Symbolic Solution or Roots . . . 3
4.1.1 The Maxima Function solve . . . 3
4.1.2 solve with Expressions or Functions & the multiplicities List . . . 4
4.1.3 General Quadratic Equation or Function . . . 5
4.1.4 Checking Solutions with subst or ev and a "Do Loop" . . . 6
4.1.5 The One Argument Form of solve . . . 7
4.1.6 Using disp, display, and print . . . 7
4.1.7 Checking Solutions using map . . . 8
4.1.8 Psuedo-PostFix Code: %% . . . 9
4.1.9 Using an Expression Rather than a Function with Solve . . . 9
4.1.10 Escape Speed from the Earth . . . 11
4.1.11 Cubic Equation or Expression . . . 14
4.1.12 Trigonometric Equation or Expression . . . 14
4.1.13 Equation or Expression Containing Logarithmic Functions . . . 15
4.2 One Equation Numerical Solutions: allroots, realroots, find_root . . . 16
4.2.1 Comparison of realroots with allroots . . . 17
4.2.2 Intersection Points of Two Polynomials . . . 18
4.2.3 Transcendental Equations and Roots: find_root . . . 21
4.2.4 find_root: Quote that Function! . . . 23
4.2.5 newton . . . 26
4.3 Two or More Equations: Symbolic and Numerical Solutions . . . 28
4.3.1 Numerical or Symbolic Linear Equations with solve or linsolve . . . 28
4.3.2 Matrix Methods for Linear Equation Sets: linsolve_by_lu . . . 29
4.3.3 Symbolic Linear Equation Solutions: Matrix Methods . . . 30
4.3.4 Multiple Solutions from Multiple Right Hand Sides . . . 31
4.3.5 Three Linear Equation Example . . . 32
4.3.6 Surpressing rat Messages: ratprint . . . 34
4.3.7 Non-Linear Polynomial Equations . . . 35
4.3.8 General Sets of Nonlinear Equations: eliminate, mnewton . . . 37
4.3.9 Intersections of Two Circles: implicit_plot . . . 37
4.3.10 Using Draw for Implicit Plots . . . 38
4.3.11 Another Example . . . 39
4.3.12 Error Messages and Do It Yourself Mnewton . . . 42
4.3.13 Automated Code for mymnewton . . . 45

*This version uses Maxima 5.17.1. This is a live document. Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions to `woollett@charter.net`

COPYING AND DISTRIBUTION POLICY

This document is part of a series of notes titled ”Maxima by Example” and is made available via the author’s webpage `http://www.csulb.edu/˜woollett/` to aid new users of the Maxima computer algebra system.

NON-PROFIT PRINTING AND DISTRIBUTION IS PERMITTED.
You may make copies of this document and distribute them to others as long as you charge no more than the costs of printing.

These notes (with some modifications) will be published in book form eventually via Lulu.com in an arrangement which will continue to allow unlimited free download of the pdf files as well as the option of ordering a low cost paperbound version of these notes.

4 Solving Equations

Maxima has several functions which can be used for solving sets of algebraic equations and for finding the roots of an expression. These are described in the Maxima manual, Sec. 21, and listed under ”Contents” under ”Equations”.

This chapter gives examples of the following Maxima functions:

- **solve** solves a system of simultaneous linear or nonlinear polynomial equations for the specified variable(s) and returns a list of the solutions.
- **linsolve** solves a system of simultaneous linear equations for the specified variables and returns a list of the solutions.
- **find_root** uses a combination of binary search and Newton-Raphson methods for univariate functions and will find a root when provided with an interval containing at least one root.
- **allroots** finds all the real and complex roots of a real univariate polynomial.
- **realroots** finds all of the real roots of a univariate polynomial within a specified tolerance.
- **eliminate** eliminates variables from a set of equations or expressions.
- **linsolve_by_lu** solves a system of linear algebraic equations by the matrix method known as ”LU decomposition”, and provides a Maxima method to work with a set of linear equations in terms of the matrix of coefficients.
- **newton**, naive univariate Newton-Raphson, and **mnewton**, multivariate Newton-Raphson, can deal with nonlinear function(s).

We also encourage the use of two dimensional plots to approximately locate solutions.

This chapter does not yet include ”Solving Recurrence Relations”, and ”Solving One Hundred Equations”.

4.1 One Equation or Expression: Symbolic Solution or Roots

4.1.1 The Maxima Function solve

Maxima's ability to solve equations is limited, but progress is being made in this area. The Maxima manual has an extensive entry for the important function **solve**, which you can view in Maxima with the input `? solve` (no semicolon) followed by (Enter), or the equivalent command: `describe(solve)$`. The input `example(solve)$` will show you the manual examples without the manual syntax material. We will present some examples of the use of **solve** and not try to cover ”everything”.
solve tries to find exact solutions. If **solve**`(f(x),x)` cannot find an exact solution, **solve** tries to return a simplified version of the original problem. Sometimes the ”simplified” version can be useful:

```
(%i1) f(x);
(%o1)                                    f(x)
(%i2) solve( f(x)^2-1 , x );
(%o2)                          [f(x) = - 1, f(x) = 1]
```

Since Maxima's idea of what is ”simpler” may not agree with your own, often the returned version is of no use.

The Maxima manual **solve** syntax discussion relevant to solving one equation is:

> Function: **solve** `(expr, x)`
> Function: **solve** `(expr)`
> Solves the algebraic equation `expr` for the variable `x` and returns a list of solution equations in `x`. If `expr` is not an equation, the equation `expr = 0` is assumed in its place. `x` may be a function (e.g. `f(x)`), or other non-atomic expression except a sum or product. `x` may be omitted if `expr` contains only one variable. `expr` may be a rational expression, and may contain trigonometric functions, exponentials, etc.
> `breakup` if `false` will cause `solve` to express the solutions of cubic or quartic equations as single expressions rather than as made up of several common subexpressions which is the default.
> `multiplicities` will be set to a list of the multiplicities of the individual solutions returned by `solve`, `realroots`, or `allroots`.
> Try `apropos (solve)` for the switches which affect `solve`. `describe` may then by used on the individual switch names if their purpose is not clear.

It is important to recognise that the first argument to **solve** is either an equation such as `f(x) = g(x)` (or `h(x) = 0`), or simply `h(x)`; in the latter case, **solve** understands that you mean the equation `h(x) = 0`, and the problem is to find the "roots" of `h(x)`, ie., values of `x` such that the equation `h(x) = 0` is satisfied.

Here we follow the manual suggestion about using **apropos** and **describe**:

```
(%i1) apropos(solve);
(%o1) [solve, solvedecomposes, solveexplicit, solvefactors, solvenullwarn,
                       solveradcan, solvetrigwarn, solve_inconsistent_error]
(%i2) describe(solveradcan)$
 -- Option variable: solveradcan
     Default value: `false'

     When `solveradcan' is `true', `solve' calls `radcan' which makes
     `solve' slower but will allow certain problems containing
     exponentials and logarithms to be solved.
(%i3) describe(solvetrigwarn)$
 -- Option variable: solvetrigwarn
     Default value: `true'

     When `solvetrigwarn' is `true', `solve' may print a message saying
     that it is using inverse trigonometric functions to solve the
     equation, and thereby losing solutions.
```

4.1.2 solve with Expressions or Functions & the multiplicities List

Let's start with a simple example where the expected answers are obvious and check the behavior of **solve**. In particular we want to check **solve**'s behavior with both an expression and a function (defined via `:=`). We also want to check how the system list **multiplicities** is created and maintained. We include the use of **realroots** and **allroots** in this comparison, even though we will not have to use these latter two functions for a while.

```
(%i1) multiplicities;
(%o1)                             not_set_yet
(%i2) ex1 : x^2 - 2*x + 1;
                                    2
(%o2)                              x  - 2 x + 1
```

```
(%i3) factor(ex1);
                                              2
(%o3)                                  (x - 1)
(%i4) g(x) := x^2 - 2*x + 1$
(%i5) g(y);
                                      2
(%o5)                                y  - 2 y + 1
(%i6) solve(ex1);
(%o6)                                   [x = 1]
(%i7) multiplicities;
(%o7)                                     [2]
(%i8) solve(g(y));
(%o8)                                   [y = 1]
(%i9) multiplicities;
(%o9)                                     [2]
(%i10) realroots(ex1);
(%o10)                                  [x = 1]
(%i11) multiplicities;
(%o11)                                    [2]
(%i12) allroots(ex1);
(%o12)                            [x = 1.0, x = 1.0]
(%i13) multiplicities;
(%o13)                                    [2]
```

We see that we can use either an expression or a function with **solve**, and you can check that this also applies to **realroots** and **allroots**. It is not clear from our use of **allroots** above how **allroots** affects **multiplicities**, although, as we will see later, the manual does not assert any connection, and we would not expect there to be a connection because **allroots** returns multiple roots explicitly in `%o12`. Just to make sure, let's restart Maxima and use only **allroots**:

```
(%i1) multiplicities;
(%o1)                                not_set_yet
(%i2) allroots(x^2 - 2*x + 1);
(%o2)                            [x = 1.0, x = 1.0]
(%i3) multiplicities;
(%o3)                                not_set_yet
```

As we expected, **allroots** does not affect **multiplicities**; only **solve** and **realroots** set its value.

4.1.3 General Quadratic Equation or Function

To get our feet wet, lets turn on the machinery with a general quadratic equation or expression. There are some differences if you employ an expression rather than a function defined with `:=`. Each method has some advantages and some disadvantages. Let's first use the function argument, rather than an expression argument. We will later show how the calculation is different if an expression is used. We will step through the process of verifying the solutions and end up with a "do loop" which will check all the solutions. We will use a function `f(x)` which depends parametrically on `(a,b,c)` as the first argument to **solve**, and first see what happens if we don't identify the unknown: how smart is Maxima??

```
(%i1) f(x) := a*x^2 + b*x + c$
```

```
(%i2) f(y);
                                         2
(%o2)                                 a y  + b y + c
(%i3) sol : solve( f(x) );
More unknowns than equations - `solve'
Unknowns given :
[a, x, b, c]
Equations given:
    2
[a x  + b x + c]
 -- an error.  To debug this try debugmode(true);
```

We see that Maxima cannot read our mind! We must tell Maxima which of the four symbols is to be considered the "unknown". From Maxima's point of view (actually the point of view of the person who wrote the code), one equation cannot determine four unknowns, so we must supply the information about which of the four variables is to be considered the unknown.

```
(%i4) sol : solve( f(x),x );
                              2                          2
                    sqrt(b  - 4 a c) + b       sqrt(b  - 4 a c) - b
(%o4)        [x = - --------------------, x = --------------------]
                             2 a                        2 a
```

We see that **solve** returns the expected list of two possible symbolic solutions.

4.1.4 Checking Solutions with subst or ev and a "Do Loop"

Let's check the first solution:

```
(%i5) s1 : sol[1];
                                            2
                                     sqrt(b  - 4 a c) + b
(%o5)                          x = - --------------------
                                              2 a
```

Now we can use the `subst( x = x1, f(x) )` form of the **subst** function syntax.

```
(%i6) r1 : subst(s1, f(x) );
                     2               2             2
            (sqrt(b  - 4 a c) + b)    b (sqrt(b  - 4 a c) + b)
(%o6)       ---------------------- - ------------------------ + c
                       4 a                        2 a
(%i7) expand(r1);
(%o7)                                  0
```

Now that we understand what steps lead to the desired "0", we automate the process using a do loop:

```
(%i8) for i:1 thru 2 do disp( expand( subst( sol[i], f(x) ) ) )$
                                       0
                                       0
```

For each of the two solutions (for `x`) found by Maxima, the given expression evaluates to zero, verifying the roots of the expression.

Since the result (here) of using `ev( f(x), sol[i])` is the same as using `subst( sol[i], f(x) )`, we can use **ev** instead:

```
(%i9) for i:1 thru 2 do disp( expand( ev(f(x), sol[i]) ))$
                                        0
                                        0
```

4.1.5 The One Argument Form of solve

The simple one-argument form of **solve** can be used if all but one of the symbols in the expression is already "bound".

```
(%i10) solve(3*x -2);
                                          2
(%o10)                               [x = -]
                                          3
(%i11) (a:1, b:2, c:3)$
(%i12) [a,b,c];
(%o12)                              [1, 2, 3]
(%i13) solve(a*x^2 + b*x + c);
(%o13)            [x = - sqrt(2) %i - 1, x = sqrt(2) %i - 1]
(%i14) [a,b,c] : [4,5,6];
(%o14)                              [4, 5, 6]
(%i15) solve(a*x^2 + b*x + c);
                        sqrt(71) %i + 5      sqrt(71) %i - 5
(%o15)           [x = - ---------------, x = ---------------]
                               8                    8
(%i16) [a,b,c] : [4,5,6];
(%o16)                               [4, 5, 6]
```

4.1.6 Using disp, display, and print

We have seen above examples of using **disp**, which can be used to print out the values of symbols or text, and **display**, which can be used to print out the name of the symbol and its value in the form of an equation: "x = value".
Here is the do loop check of the roots of the quadratic found above using **print** instead of **disp**.
However, we need to be careful, because we are using a function `f(x)` rather than an expression. We have just assigned the values of `a`, `b`, and `c`, and we want `f(x)` to have arbitrary values of these parameters.

```
(%i17) [a,b,c];
(%o17)                              [4, 5, 6]
(%i18) f(x);
                                   2
(%o18)                          4 x  + 5 x + 6
(%i19) kill(a,b,c);
(%o19)                                done
(%i20) [a,b,c];
(%o20)                              [a, b, c]
(%i21) f(x);
                                   2
(%o21)                          a x  + b x + c
```

```
(%i22) sol;
                          2                       2
                 sqrt(b  - 4 a c) + b      sqrt(b  - 4 a c) - b
(%o22)      [x = - --------------------, x = --------------------]
                          2 a                      2 a
(%i23) for i:1 thru 2 do print("expr = ", expand( subst(sol[i],f(x) ) ) )$
expr =  0
expr =  0
```

Here we use **disp** to display a title for the do loop:

```
(%i24) ( disp("check roots"), for i thru 2 do
               print("expr = ", expand( subst( sol[i],f(x) ) ) )  )$
                                  check roots

expr =  0
expr =  0
```

The only tricky thing about this kind of code is getting the parentheses to balance. Note that that **expand(...)** is inside **print**, so the syntax used is **do print(...)**, ie., a ”one job do ”. The outside parentheses allow the syntax `( job1, job2 )`. Note also that the default start of the do loop index is ”1”, so we can use an abbreviated syntax that does not have the `i:1` beginning.

4.1.7 Checking Solutions using map

One advantage of using a function `f(x)` defined via `:=` as the first argument to **solve** is that it is fairly easy to check the roots by using the **map** function. We want to use the syntax `map( f, solnlist}`, where `solnlist` is a list of the roots (not a list of replacement rules). To get the solution list we can again use **map** with the syntax `map(rhs, sol}`.

```
(%i25) solnlist : map( rhs, sol );
                          2                    2
                 sqrt(b  - 4 a c) + b  sqrt(b  - 4 a c) - b
(%o25)          [- --------------------, --------------------]
                          2 a                    2 a
(%i26) map( f, solnlist );
                 2              2            2
        (sqrt(b  - 4 a c) + b)     b (sqrt(b  - 4 a c) + b)
(%o26) [---------------------- - ------------------------ + c,
                 4 a                         2 a
                                  2              2            2
                         (sqrt(b  - 4 a c) - b)     b (sqrt(b  - 4 a c) - b)
                         ---------------------- + ------------------------ + c]
                                  4 a                         2 a
(%i27) expand(%);
(%o27)                               [0, 0]
(%i28) expand( map(f, map(rhs, sol) ) );
(%o28)                               [0, 0]
```

The last input `%i27` shows a compact method which avoids having to name the ”solnlist” and which also avoids having to look at the intermediate output. When you see someone’s example written in a compact form like this, you should realize that the ”someone” probably tried out the progression of steps one step at a time (just like we did) to see the correct route, and once the path to the result has been found, reduced the result to

the minimum number of steps and names. Often, one does not know in advance which progression of steps will succeed, and one must experiment before finding the "true path". You should "take apart" the compact code, by reading from the inside out (ie., from right to left), and also try getting the result one step at a time to get comfortable with the method and notation being used.

4.1.8 Psuedo-PostFix Code: %%

An alternative "psuedo-postfix" (ppf) notation can be used which allows one to read the line from left to right, following the logical succession of procedures being used. Although this ppf notation costs more in keystrokes (an extra pair of outside parentheses, extra commas, and entry of double percent signs `%%`), the resulting code is usually easier for beginners to follow, and it is easier to mentally balance parentheses as well. As an example, the previous double map check of the roots can be carried out as:

```
(%i29) ( map(rhs,sol), map(f,%%), expand(%%) );
(%o29)                               [0, 0]
```

Note the beginning and ending parentheses for the whole "line" of input, with the syntax:
`( job1, job2(%%), job3(%%),... )`. The system variable `%%` has the manual description (in part):

> System variable: `%%`
> In compound statements, namely `block`, `lambda`, or `(s_1, ..., s_n)`, `%%` is the value of the previous statement.

4.1.9 Using an Expression Rather than a Function with Solve

Let's rework the general quadratic equation solution, including the checks of the solutions, using an expression `ex` rather than a function `f(x)` defined using `:=`.

```
(%i1) ex : a*x^2 + b*x + c$
(%i2) sol : solve( ex, x );
                        2                           2
                sqrt(b  - 4 a c) + b       sqrt(b  - 4 a c) - b
(%o2)        [x = - --------------------, x = --------------------]
                           2 a                         2 a
(%i3) s1 : sol[1];
                                       2
                                sqrt(b  - 4 a c) + b
(%o3)                        x = - --------------------
                                          2 a
(%i4) r1 : subst(s1, ex );
                  2              2            2
           (sqrt(b  - 4 a c) + b)     b (sqrt(b  - 4 a c) + b)
(%o4)      ----------------------- - ------------------------ + c
                    4 a                         2 a
(%i5) expand(r1);
(%o5)                                 0
(%i6) for i:1 thru 2 do disp( expand( subst( sol[i], ex ) ) )$
                                      0
                                      0
```

(We could have also used **ev** instead of **subst**.)

Thus far, the methods have been similar. If we now bind the values of `(a,b,c)`, as we did in the middle of our solutions using `f(x)`, what is the difference?

```
(%i7) [a,b,c] : [1,2,3]$
(%i8) [a,b,c];
(%o8)                          [1, 2, 3]
(%i9) ex;
                                 2
(%o9)                         a x  + b x + c
```

We see that the symbol `ex` remains bound to the same general expression. The symbol `ex` retains its original binding. We can make use of the values given to `(a,b,c)` with the expression `ex` by using two single quotes, which forces an extra evaluation of the expression `ex` by the Maxima engine, and which then makes use of the extra information about `(a,b,c)`.

```
(%i10) ''ex;
                                2
(%o10)                         x  + 2 x + 3
(%i11) ex;
                                 2
(%o11)                        a x  + b x + c
```

Forcing the extra evaluation in `%10` does not change the binding of `ex`. Now let's try to check the solutions using **map**, as we did before. To use **map** we need a function, rather than an expression to map on a solution list. Let's try to define such a function `f(x)` using the expression `ex`.

```
(%i12) f(x);
(%o12)                            f(x)
(%i13) f(x) := ex;
(%o13)                         f(x) := ex
(%i14) f(y);
                                 2
(%o14)                        a x  + b x + c
(%i15) f(x) := ''ex;
                                     2
(%o15)                   f(x) := a x  + b x + c
(%i16) f(y);
                                2
(%o16)                         y  + 2 y + 3
(%i17) kill(a,b,c);
(%o17)                            done
(%i18) f(y);
                                 2
(%o18)                        a y  + b y + c
(%i19) solnlist : map(rhs,sol);
                       2                     2
                 sqrt(b  - 4 a c) + b  sqrt(b  - 4 a c) - b
(%o19)        [- --------------------, --------------------]
                         2 a                    2 a
```

```
(%i20) map(f,solnlist);
               2              2            2
        (sqrt(b  - 4 a c) + b)     b (sqrt(b  - 4 a c) + b)
(%o20) [----------------------- - ------------------------ + c,
                 4 a                         2 a
                                   2              2            2
                            (sqrt(b  - 4 a c) - b)     b (sqrt(b  - 4 a c) - b)
                            ----------------------- + ------------------------ + c]
                                     4 a                         2 a
(%i21) expand(%);
(%o21)                               [0, 0]
```

Output `%14` showed that the syntax `f(x) := ex` did not succeed in defining the function we need. The input `f(x) := ''ex` suceeded in getting a true function of `x`, but now the function `f(x)` automatically makes use of the current binding of `(a,b,c)`, so we had to **kill** those values to get a function with arbitrary values of `(a,b,c)`. Having the function in hand, we again used the **map** function twice to check the solutions. Now that we have discovered the "true path", we can restart Maxima and present the method as:

```
(%i1) ex : a*x^2 + b*x + c$
(%i2) sol : solve( ex, x );
                            2                          2
                    sqrt(b  - 4 a c) + b      sqrt(b  - 4 a c) - b
(%o2)       [x = - --------------------, x = --------------------]
                           2 a                        2 a
(%i3) f(x) := ''ex$
(%i4) expand ( map(f, map(rhs, sol) ) );
(%o4)                               [0, 0]
```

We can also use the (generally safer) syntax `define( f(x),ex );` to obtain a true function of `x`:

```
(%i5) define( f(x),ex );
                                           2
(%o5)                       f(x) := a x  + b x + c
(%i6) f(y);
                                       2
(%o6)                           a y  + b y + c
(%i7) expand ( map(f, map(rhs, sol) ) );
(%o7)                               [0, 0]
```

We can also use the unnamed, anonymous function **lambda** to avoid introducing needless names, like "`f`":

```
(%i8) expand( map(lambda([x],''ex), map(rhs,sol) ) );
(%o8)                               [0, 0]
```

4.1.10 Escape Speed from the Earth

In this section we solve a physics problem which involves a simple quadratic equation. It is so simple that "doing it" on paper is faster than doing it with Maxima. In fact, once you understand the plan of the calculation, you can come up with the final formula for the escape speed in your head. However, we will present practical details of setup and evaluation which can be used with more messy problems, when you might want to use Maxima.

Let's use conservation of mechanical energy (kinetic plus potential) to first calculate the initial radial speed a rocket must have near the surface of the earth to achieve a final required radial speed far from the earth (far

enough away so we can neglect earth's gravitational pull).
Let the mass of the rocket be `m`, the mass of the earth be `M`, the radius of the earth be `R`, a general radial distance from the center of the earth be `r >= R`, a general radial rocket speed be `v`, the maximum speed of the rocket near the surface of the earth be `v0`, and the final radial speed of the rocket (as `r` becomes infinite) be `vf`.

At a general distance `r` from the center of the earth the rocket has kinetic energy `ke = m*v^2/2`, and gravitational energy `pe = -G*M*m/r`, where `G` is the gravitational constant:
`(G = 6.673 10^(-11) newton*meter^2/kg^2)`.

```
(%i1) energy : m*v^2/2 - G*M*m/r;
                                       2
                                    m v     m G M
(%o1)                               ---- - -----
                                     2       r
```

The initial energy `e0` corresponds to the energy the rocket has achieved at the moment of maximum radial speed: this will occur at a radius `r` slightly larger than the radius of the earth `R`, but negligible error to the required "lift-off speed" `v0` will be made by ignoring this difference in radius. (You can justify this as a good approximation by getting the answer when including this small difference, and comparing the percent difference in the answers.)

```
(%i2) e0 : energy,v=v0,r=R;
                                        2
                                    m v0     m G M
(%o2)                               ----- - -----
                                      2       R
```

As the rocket "rises", `r` becomes larger, and the magnitude of the gravitational energy becomes smaller. The "final" energy `efinal` will be the energy when the gravitational energy is so small that we can ignore it; in practice this will occur when the magnitude of the gravitational energy is much smaller than the magnitude of the inital gravitational energy. The radial outward speed of the rocket then remains a constant value `vf`.

```
(%i3) efinal : limit(energy,r,inf),v=vf;
                                            2
                                        m vf
(%o3)                                   -----
                                          2
```

If we neglect the loss of mechanical energy due to friction in leaving the earth's atmosphere, and also neglect other tiny effects like the gravitational interaction between the moon and the rocket, the sun and the rocket, etc, then we can approximately say that the total mechanical energy (as we have defined it) of the rocket is a constant, once chemical energy used to increase the rocket's speed is no longer a factor (which occurs at the moment of maximum radial speed).
We can then get one equation by approximately equating the mechanical energy of the rocket just after achieving maximum speed to the mechanical energy of the rocket when `r` is so large that we can ignore the instantaneous gravitational energy contribution.

```
(%i4) v0soln : solve(efinal = e0,v0);
                           2 G M     2                2 G M     2
(%o4)         [v0 = - sqrt(----- + vf ), v0 = sqrt(----- + vf )]
                             R                        R
```

```
(%i5) v0soln : v0soln[2];
                                        2 G M     2
(%o5)                       v0 = sqrt(----- + vf )
                                          R
(%i6) v0;
(%o6)                                   v0
(%i7) v0 : rhs( v0soln );
                                      2 G M     2
(%o7)                          sqrt(----- + vf )
                                        R
```

This provides the required "lift-off" speed near the surface of the earth to achieve a given final radial speed `vf`. We now want to find the "escape speed", the minimum value of the lift-off speed which will allow the rocket to escape the gravitational pull of the earth. Any rocket which has a radial speed when `r` is effectively infinite will succeed in escaping, no matter how small that radial speed is. The limiting initial speed is then gotten by taking the limit of `v0` as `vf` goes to zero.

```
(%i8) vescape : ev( v0, vf = 0 );
                                            G M
(%o8)                          sqrt(2) sqrt(---)
                                             R
```

Uing the mass of the earth, `M = 5.974 10^(24) kg`, and the radius of the earth, `R = 6.378 10^(6) meters`, we get an escape speed `11,181 m/s = 11.18 km/s`.

```
(%i09) ev( vescape, [G=6.673e-11,M=5.974e24,R=6.378e6] );
(%o09)                    7905.892670345354 sqrt(2)
(%i10) float(%);
(%o10)                        11180.62063706845
```

We have rounded the answer to four significant figures, since that is the accuracy of the Earth data and gravitational constant we have used.
If we need to use a set of physical or numerical constants throughout a session, we can define a list of "equalities", say `clist`, and use as follows:

```
(%i11) clist : [G=6.673e-11,M=5.974e24,R=6.378e6];
(%o11) [G = 6.672999999999999E-11, M = 5.9740000000000004E+24, R = 6378000.0]
(%i12) ev( vescape, clist, float );
(%o12)                         11180.62063706845
```

Note that adding the option variable **float** as a switch (equivalent to `float:true`), gets the `sqrt(2)` in floating point.

Looking at all those digits is unnecessary; set **fpprintprec** to something reasonable (this only affects the numbers presented on the screen, not the accuracy of the calculation):

```
(%i13) fpprintprec:8$
(%i14) ev( vescape, clist, float );
(%o14)                              11180.621
(%i15) clist;
(%o15)            [G = 6.673E-11, M = 5.974E+24, R = 6378000.0]
```

4.1.11 Cubic Equation or Expression

Here is an example of using **solve** to "solve" a cubic equation, or, in the alternative language, find the roots of a cubic expression. After checking the roots via the **map** function, we assign the values of the roots to the symbols `(x1,x2,x3)`. The cubic expression we choose is especially simple, with no arbitrary parameters, so we can use the one argument form of **solve**.

```
(%i1) ex : x^3 + x^2 + x$
(%i2) sol : solve(ex);
                      sqrt(3) %i + 1      sqrt(3) %i - 1
(%o2)         [x = - --------------, x = --------------, x = 0]
                            2                   2
(%i3) define( f(x), ex )$
(%i4) expand ( map(f, map(rhs, sol) ) );
(%o4)                               [0, 0, 0]
(%i5) [x1,x2,x3] : map(rhs,sol);
                        sqrt(3) %i + 1  sqrt(3) %i - 1
(%o5)                 [- --------------, --------------, 0]
                              2               2
(%i6) x1;
                                 sqrt(3) %i + 1
(%o6)                          - --------------
                                       2
```

4.1.12 Trigonometric Equation or Expression

Here is an exact solution using **solve**:

```
(%i1) [fpprintprec:8,display2d:false]$
(%i2) ex : sin(x)^2 -2*sin(x) -3$
(%i3) sol : solve(ex);
`solve' is using arc-trig functions to get a solution.
Some solutions will be lost.
(%o3) [x = asin(3),x = -%pi/2]
(%i4) define( f(x), ex )$
(%i5) expand ( map(f, map(rhs, sol) ) );
(%o5) [0,0]
(%i6) numroots : float( map(rhs, sol) );
(%o6) [1.5707963-1.7627472*%i,-1.5707963]
```

The first solution returned is the angle (in radians) whose sin is 3. For real `x`, `sin(x)` lies in the range `-1 <=  sin(x)  <= 1` . Thus we have found one real root. But we have been warned that some solutions will be lost. Because the given expression is a polynomial in `sin(x)`, we can use **realroots**:

```
(%i7) rr : realroots(ex);
(%o7) [sin(x) = -1,sin(x) = 3]
```

However, by "realroots", **realroots** means that the numbers `[-1,3]` are real!

We can of course take the output of **realroots** and let **solve** go to work.

```
(%i8) map(solve, rr);
`solve' is using arc-trig functions to get a solution.
Some solutions will be lost.

`solve' is using arc-trig functions to get a solution.
Some solutions will be lost.
(%o8) [[x = -%pi/2],[x = asin(3)]]
```

We know that the numerical value of the expression `ex3` repeats when `x` is replaced by `x + 2*%pi`, so there are an infinite number of real roots, related to $-\pi/2$ by adding or subtracting $2n\pi$, where n is an integer.

We can make a simple plot of our expression to see the periodic behavior and the approximate location of the real roots.

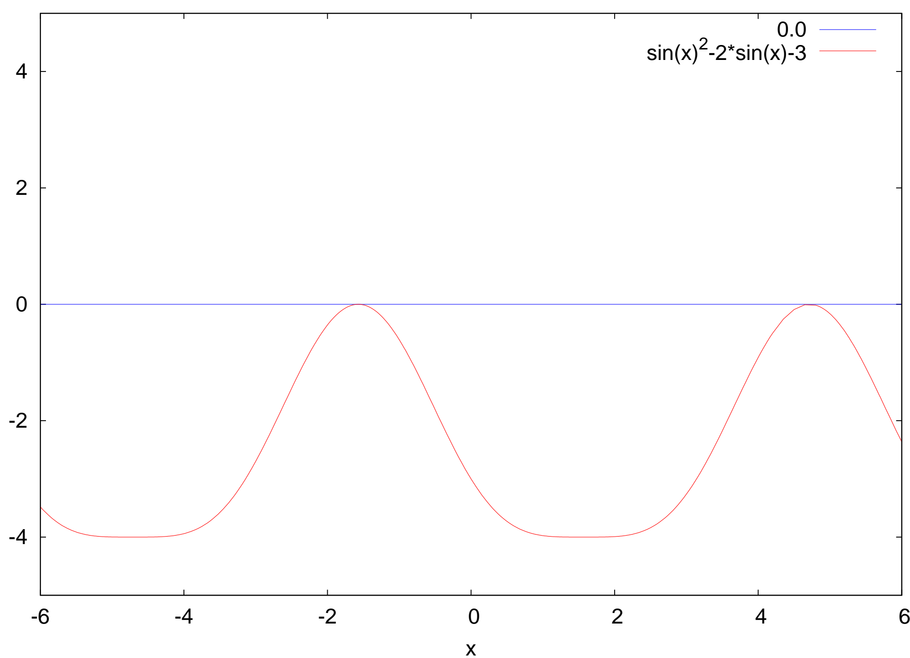

Figure 1: plot of ex3

We used the **plot2d** code:

```
(%i18) plot2d([0.0,ex3],[x,-6,6],[y,-5,5] )$
```

4.1.13 Equation or Expression Containing Logarithmic Functions

Here is an example submitted to the Maxima mailing list and a method of solution provided by Maxima developer Stavros Macrakis. The problem is to find the roots of the following expression `ex`:

```
(%i1) [fpprintprec:8,display2d:false,ratprint:false]$
(%i2) ex : log(0.25*(2*x+5)) - 0.5*log(5*x - 5)$
```

We first try **solve**, with the option variable **solveradcan** set equal to **true**. Remember that the syntax `func, optvar;` is equivalent to `func, optvar:true;`.

```
(%i3) sol : solve(ex,x),solveradcan;
(%o3) [log((2*x+5)/4) = log(5*x-5)/2]
```

We see that **solve** tried to find a "simpler" form which it returned in `%o2`. The Maxima function **fullratsimp** has the manual description

> Function: **fullratsimp** `(expr)`
> `fullratsimp` repeatedly applies `ratsimp` followed by non-rational simplification to an expression until no further change occurs, and returns the result.
> When non-rational expressions are involved, one call to `ratsimp` followed as is usual by non-rational ("general") simplification may not be sufficient to return a simplified result. Sometimes, more than one such call may be necessary. `fullratsimp` makes this process convenient.

The effect of **fullratsimp** in our case results in the decimals being replaced by exact fractions.

```
(%i4) ex : fullratsimp(ex);
(%o4) (2*log((2*x+5)/4)-log(5*x-5))/2
```

The logarithms can be combined using the Maxima function **logcontract**. This function was discussed in Chapter 2, Sect. 2.3.5. A partial description is:

> Function: **logcontract** `(expr)`
> Recursively scans the expression `expr`, transforming subexpressions of the form
> `a1*log(b1) + a2*log(b2) + c` into `log(ratsimp(b1^a1 * b2^a2)) + c`

```
(%i1) 2*(a*log(x) + 2*a*log(y))$
(%i2) logcontract(%);
                                       2  4
(%o2)                       a log(x   y )
```

Here is the application to our problem:

```
(%i5) ex : logcontract(ex);
(%o5) -log((80*x-80)/(4*x^2+20*x+25))/2
```

Having combined the logarithms, we try out **solve** on this expression:

```
(%i6) sol : solve(ex );
(%o6) [x = -(2*sqrt(30)-15)/2,x = (2*sqrt(30)+15)/2]
```

We have a successful exact solution, but **solve** (in its present incarnation) needed some help. We now use the **map** method to check the roots.

```
(%i7) define( f(x), ex )$
(%i8) expand( map(f, map(rhs,sol) ) );
(%o8) [0,0]
```

4.2 One Equation Numerical Solutions: allroots, realroots, find_root

We have already tried out the Maxima functions **realroots** and **allroots**. The most important restriction for both of these numerical methods is that the expression or equation be a polynomial, as the manual explains:

> – Function: **realroots** `(eqn, bound)`
> – Function: **realroots** `(expr)`
> – Function: **realroots** `(eqn)`
> Computes rational approximations of the real roots of the polynomial `expr` or polynomial equation `eqn` of one variable, to within a tolerance of `bound`. Coefficients of `expr` or `eqn` must be literal numbers; symbol constants such as `%pi` are rejected.
> `realroots` assigns the multiplicities of the roots it finds to the global variable `multiplicities`.

realroots constructs a Sturm sequence to bracket each root, and then applies bisection to refine the approximations. All coefficients are converted to rational equivalents before searching for roots, and computations are carried out by exact rational arithmetic. Even if some coefficients are floating-point numbers, the results are rational (unless coerced to floats by the float or numer flags).
When bound is less than 1, all integer roots are found exactly. When bound is unspecified, it is assumed equal to the global variable rootsepsilon (default: 10^(-7)).
When the global variable programmode is true (default: true), realroots returns a list of the form [x = <x_1>, x = <x_2>, ...]. When programmode is false, realroots creates intermediate expression labels %t1, %t2, ..., assigns the results to them, and returns the list of labels.

Here are the startup values of the option variables just mentioned:

```
(%i1) fpprintprec:8$
(%i2) [multiplicities,rootsepsilon,programmode];
(%o2)                    [not_set_yet, 1.0E-7, true]
```

The function **allroots** also accepts only polynomials, and finds numerical approximations to both real and complex roots:

Function: **allroots** (expr)
Function: **allroots** (eqn)
Computes numerical approximations of the real and complex roots of the polynomial expr or polynomial equation eqn of one variable.
The flag polyfactor when true causes allroots to factor the polynomial over the real numbers if the polynomial is real, or over the complex numbers, if the polynomial is complex (default setting of polyfactor is false).
allroots may give inaccurate results in case of multiple roots.
If the polynomial is real, allroots (%i*p)) may yield more accurate approximations than allroots(p), as allroots invokes a different algorithm in that case.
allroots rejects non-polynomials. It requires that the numerator after rat'ing should be a polynomial, and it requires that the denominator be at most a complex number. As a result of this, allroots will always return an equivalent (but factored) expression, if polyfactor is true.

Here we test the default value of **polyfactor**:

```
(%i3) polyfactor;
(%o3)                              false
```

4.2.1 Comparison of realroots with allroots

Let's find the real and complex roots of a fifth order polynomial which **solve** cannot "solve", doesn't factor, and use both **realroots** and **allroots**.

```
(%i4) ex : x^5 + x^4 -4*x^3 +2*x^2 -3*x -7$
(%i5) define( fex(x), ex )$
```

We first use **realroots** to find the three real roots of the given polynomial, and substitute the roots back into the expression to see how close to zero we get.

```
(%i6) rr : float( map(rhs, realroots(ex,1e-20) ) );
(%o6)               [- 2.7446324, - 0.880858, 1.7964505]
(%i7) frr : map( fex, rr );
(%o7)                    [0.0, - 4.4408921E-16, 0.0]
```

Next we find numerical approximations to the three real roots and the two (complex-conjugate) roots of the given polynomial, using **allroots** `( ex )` and substitute the obtained roots back into the expression to see how close to zero we get.

```
(%i8) ar1 : map(rhs, allroots( ex ) );
(%o8) [1.1999598 %i + 0.41452, 0.41452 - 1.1999598 %i, - 0.880858, 1.7964505,
                                                                    - 2.7446324]
(%i9) far1 : expand( map( fex, ar1 ) );
(%o9) [1.12132525E-14 %i + 4.4408921E-16, 4.4408921E-16 - 1.12132525E-14 %i,
                         - 1.13242749E-14, 2.48689958E-14, - 2.84217094E-14]
```

Finally, we repeat the process for the syntax **allroots** `( %i* ex )`.

```
(%i10) ar2 : map(rhs, allroots( %i*ex ) );
(%o10) [1.1999598 %i + 0.41452, - 3.60716392E-17 %i - 0.880858,
0.41452 - 1.1999598 %i, 6.20555942E-15 %i + 1.7964505,
- 6.54444294E-16 %i - 2.7446324]
(%i11) far2 : expand( map( fex, ar2 ) );
(%o11) [1.55431223E-15 %i - 1.77635684E-15, 5.61204464E-16 %i - 4.4408921E-16,
1.61204383E-13 %i + 2.26041408E-13, 2.52718112E-13 %i - 1.91846539E-13,
- 6.32553289E-14 %i - 3.97903932E-13]
(%i12) far2 - far1;
(%o12) [- 9.65894031E-15 %i - 2.22044605E-15,
1.1774457E-14 %i - 8.8817842E-16, 1.61204383E-13 %i + 2.37365683E-13,
2.52718112E-13 %i - 2.16715534E-13, - 6.32553289E-14 %i - 3.69482223E-13]
```

The three real roots of the given fifth order polynomial are found more accurately by **realroots** than by either version of **allroots**. We see that the three real roots of this fifth order polynomial are found more accurately by the syntax **allroots** `(expr)` (which was used to get `ar1`), than by the syntax **allroots** `(%i*expr)`, used to get `ar2`. We also see that the syntax **allroots** `(%i*expr)` introduced a tiny complex piece to the dominant real part. The two extra complex roots found by the first syntax (`ar1`) are the complex conjugate of each other. The two extra complex roots found by the alternative syntax (`ar2`) are also the complex conjugate of each other to within the default arithmetic accuracy being used.

4.2.2 Intersection Points of Two Polynomials

Where do the two curves $h(x) = x^3 - 8\,x^2 + 19\,x - 12$ and $k(x) = \frac{1}{2}\,x^2 - x - \frac{1}{8}$ intersect? We want approximate numerical values. We can plot the two functions together and use the cursor to read off the values of `x` for which the curves cross, and we can also find the roots numerically. We first define the curves as expressions depending on `x`, then define the difference of the expressions (`rx`) to work with using **allroots** first to see if all the (three) roots are real, and then using **realroots** just for fun, and then checking the solutions. If we are just going to compare the results with a plot, we don't need any great accuracy, so we will use the default **realroots** precision.

```
(%i1) fpprintprec : 8$
(%i2) hx : x^3 - 8*x^2 + 19*x - 12$
(%i3) kx : x^2/2 -x -1/8$
(%i4) rx : hx - kx;
                                          2
                                3    17 x            95
(%o4)                          x  - ----- + 20 x - --
                                       2              8
```

```
(%i5) factor(rx);
                              3       2
                           8 x  - 68 x  + 160 x - 95
(%o5)                      -------------------------
                                       8
(%i6) define( fr(x), rx )$
(%i7) allroots(rx);
(%o7)           [x = 0.904363, x = 2.6608754, x = 4.9347613]
(%i8) rr : float( realroots(rx) );
(%o8)           [x = 0.904363, x = 2.6608754, x = 4.9347613]
(%i9) rr : map( rhs, rr);
(%o9)                 [0.904363, 2.6608754, 4.9347613]
(%i10) map(fr, rr);
(%o10)          [2.04101367E-8, - 8.4406409E-8, 5.30320676E-8]
```

We see that the numerical solutions are "zeros" of the cubic function to within the numerical accuracy **realroots** is using. Just out of curiosity, what about exact solutions of this cubic polynomial? Use of **solve** will generate a complicated looking expression involving roots and `%i`. Let's set **display2d** to **false** so the output doesn't take up so much room on the screen.

```
(%i11) display2d : false$
(%i12) sx : solve(rx);
(%o12) [x = (-sqrt(3)*%i/2-1/2)*(3^-(3/2)*sqrt(16585)*%i/16+151/432)^(1/3)
          +49*(sqrt(3)*%i/2-1/2)/(36*(3^-(3/2)*sqrt(16585)*%i/16+151/432)
                                            ^(1/3))+17/6,
                  etc, etc ]
(%i13) sx1 : map(rhs, sx);
(%o13) [(-sqrt(3)*%i/2-1/2)*(3^-(3/2)*sqrt(16585)*%i/16+151/432)^(1/3)
         +49*(sqrt(3)*%i/2-1/2)/(36*(3^-(3/2)*sqrt(16585)*%i/16+151/432)
                                           ^(1/3))+17/6,
          etc, etc ]
```

The list `sx1` holds the exact roots of the cubic polynomial which **solve** found. We see that the form returned has explicit factors of `%i`. We already know that the roots of this polynomial are purely real. How can we get the exact roots into a form where it is "obvious" that the roots are real? The Maxima expert Alexey Beshenov (via the Maxima mailing list) suggested using **rectform**, followed by **trigsimp**. Using **rectform** gets rid of the factors of `%i`, and **trigsimp** does some trig simplification.

```
(%i14) sx2 : rectform(sx1);
(%o14) [49*(3*sqrt(3)*sin(atan(432*3^-(3/2)*sqrt(16585)/(151*16))/3)/7
         -3*cos(atan(432*3^-(3/2)*sqrt(16585)/(151*16))/3)/7)/36
         +7*sqrt(3)*sin(atan(432*3^-(3/2)*sqrt(16585)/(151*16))/3)/12
         +%i*(-7*sin(atan(432*3^-(3/2)*sqrt(16585)/(151*16))/3)/12
         +49*(3*sin(atan(432*3^-(3/2)*sqrt(16585)/(151*16))/3)/7
         +3*sqrt(3)*cos(atan(432*3^-(3/2)*sqrt(16585)/(151*16))/3)/7)/36
 -7*sqrt(3)*cos(atan(432*3^-(3/2)*sqrt(16585)/(151*16))/3)/12)
         -7*cos(atan(432*3^-(3/2)*sqrt(16585)/(151*16))/3)/12+17/6,
          etc,etc   ]
(%i15) sx3 : trigsimp(sx2);
(%o15) [ (7*sqrt(3)*sin(atan(9*sqrt(16585)/(151*sqrt(3)))/3)
         -7*cos(atan(9*sqrt(16585)/(151*sqrt(3)))/3)+17)/6,
        -(7*sqrt(3)*sin(atan(9*sqrt(16585)/(151*sqrt(3)))/3)
         +7*cos(atan(9*sqrt(16585)/(151*sqrt(3)))/3)-17)/6,
 (14*cos(atan(9*sqrt(16585)/(151*sqrt(3)))/3)+17)/6 ]
```

The quantity `sx3` is a list of the "simplified" exact roots of the cubic. Using **float** we ask for the numerical values:

```
(%i16) sx4 : float(sx3);
(%o16) [2.6608754,0.904363,4.9347613]
```

We see that the numerical values agree, although the order of the roots is different. Next we enquire whether or not the "exact roots", when substituted back into the cubic, result in "exact zeroes". Mapping the cubic onto the list of roots doesn't automatically simplify to a list of three zeroes, as we would like, although applying **float** suggests the analytic roots are correct. The combination
trigsimp(expand `( [fr(root1), fr(root2), fr(root3)] ) )` still does not provide the algebraic and trig simplification needed, but we finally get `[0.0, 0.0, 0.0]` when applying **float**.

```
(%i17) float( map(fr, sx3) );
(%o17) [-3.55271368E-15,1.88737914E-15,-1.42108547E-14]
(%i18) float( expand( map(fr, sx3) ) );
(%o18) [6.66133815E-16,-2.44249065E-15,0.0]
(%i19) float( trigsimp( expand( map(fr, sx3) ) ) );
(%o19) [0.0,0.0,0.0]
```

Let's next plot the three functions, using the expressions hx, kx, and rx (the difference).

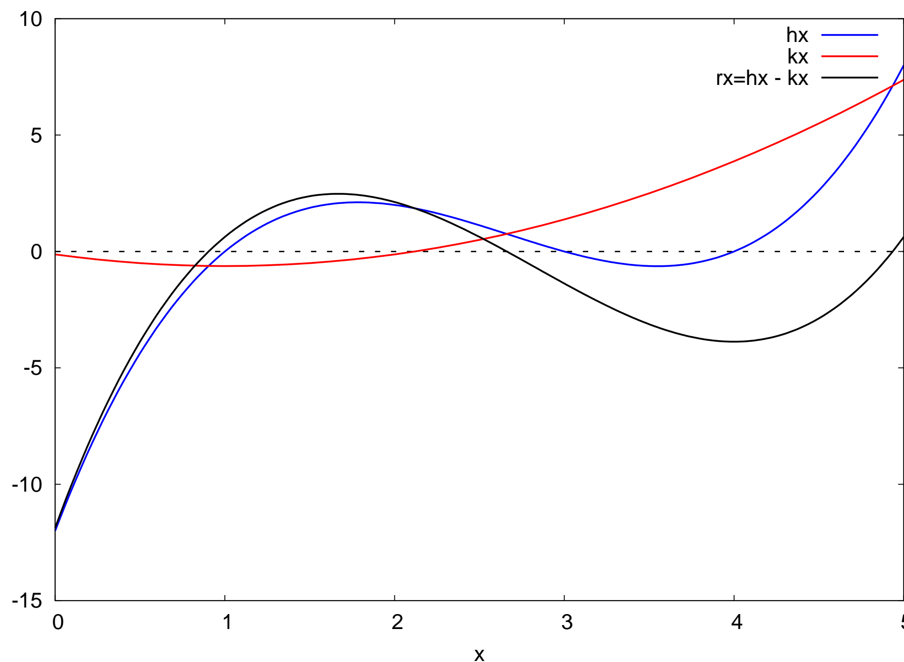

Figure 2: Intersection Points are Zeroes of rx

Here is code you can use to make something close to the above plot.

```
(%i20) plot2d([hx,kx,rx],[x,0,5],
        [style, [lines,2,1], [lines,2,2], [lines,2,0] ],
           [legend, "hx", "kx", "rx=hx - rx"],
        [gnuplot_preamble, " set xzeroaxis lw 2 "])$
```

When you place the cursor over the places on the `x` axis where the expression `rx` is zero, you can read off the coordinates in the lower left corner of the plot window. The `x` coordinate (the first) is the desired root.

4.2.3 Transcendental Equations and Roots: find_root

A transcendental equation is an equation containing a transcendental function. Examples of such equations are $x = e^x$ and $x = \sin(x)$. The logarithm and the exponential function are examples of transcendental functions. We will include the trigonometric functions, i.e., sine, cosine, tangent, cotangent, secant, and cosecant in this category of functions. (A function which is not transcendental is said to be algebraic. Examples of algebraic functions are rational functions and the square root function.)

To find the roots of transcendental expressions, for example, we can first make a plot of the expression, and then use **find_root** knowing roughly where **find_root** should start looking. The Maxima manual provides a lot of details, beginning with:

> Function: **find_root** `(expr, x, a, b)`
> Function: **find_root** `(f, a, b)`
> Option variable: **find_root_error**
> Option variable: **find_root_abs**
> Option variable: **find_root_rel**
> Finds a root of the expression `expr` or the function `f` over the closed interval `[a, b]`. The expression `expr` may be an equation, in which case `find_root` seeks a root of `lhs(expr) - rhs(expr)`. Given that Maxima can evaluate `expr` or `f` over `[a, b]` and that `expr` or `f` is continuous, `find_root` is guaranteed to find the root, or one of the roots if there is more than one.

Let's find a root of the equation $x = \cos(x)$. If we make a simple plot of the function $x - \cos(x)$, we see that there is one root somewhere between $x = 0$ and $x = 1$.

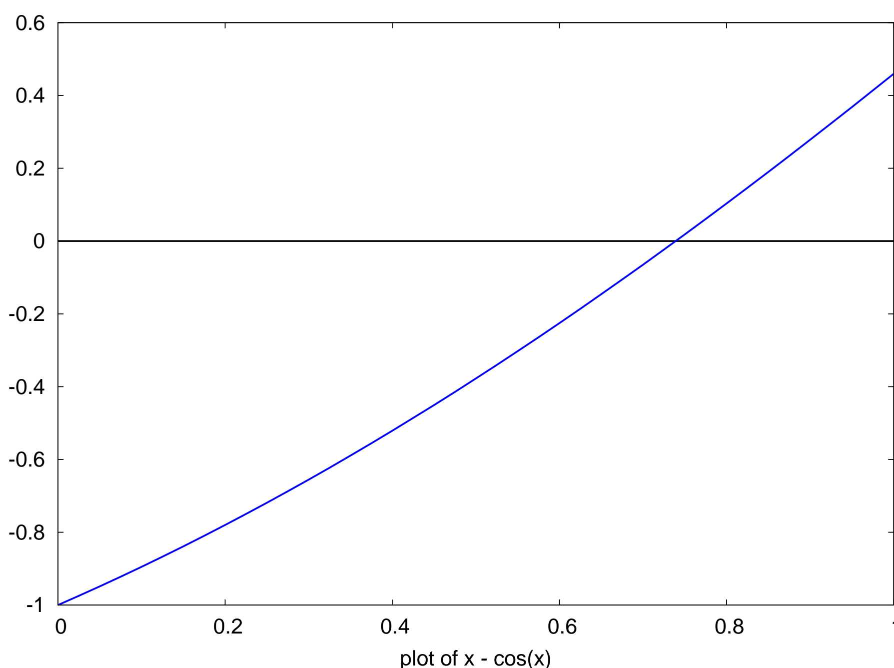

Figure 3: Plot of x - cos(x)

We can use either an expression or function as the entry to the first slot of **find_root**. I find that the most common mistake I make with **find_root** is to leave out the underline between "find" and "root", in which case, I simply get back the unevaluated "findroot(ex, x, 0, 1)", since Maxima has no knowledge of "findroot" (unless I create a homemade function with that name).

We can make a plot, find the root in a variety of ways using **find_root**, and verify the accuracy of the root as follows:

```
(%i1) fpprintprec:8$
(%i2) plot2d( x - cos(x), [ x, 0, 1 ],
                [style,  [lines, 4, 1] ],
                 [xlabel," plot of x - cos(x) "],
              [gnuplot_preamble, "set nokey; set xzeroaxis lw 2 "] )$
(%i3) find_root( x - cos(x),x, 0, 1);
(%o3)                              0.739085
(%i4) ex : x - cos(x)$
(%i5) [find_root( ex, x, 0, 1),find_root( ex, 0, 1)];
(%o5)                        [0.739085, 0.739085]
(%i6) define( f(x), ex )$
(%i7) [find_root(f(x), x, 0, 1), find_root(f(x), 0, 1),
          find_root(f, 0, 1), find_root(f, x, 0, 1)];
(%o7)             [0.739085, 0.739085, 0.739085, 0.739085]
(%i8) ev(ex, x = first(%) );
(%o8)                                0.0
```

As a second example, we will find the roots of the function

$$f(x) = \cos(x/\pi)\, e^{-(x/4)^2} - \sin(x^{3/2}) - 5/4$$

Here is the plot of $f(x)$.

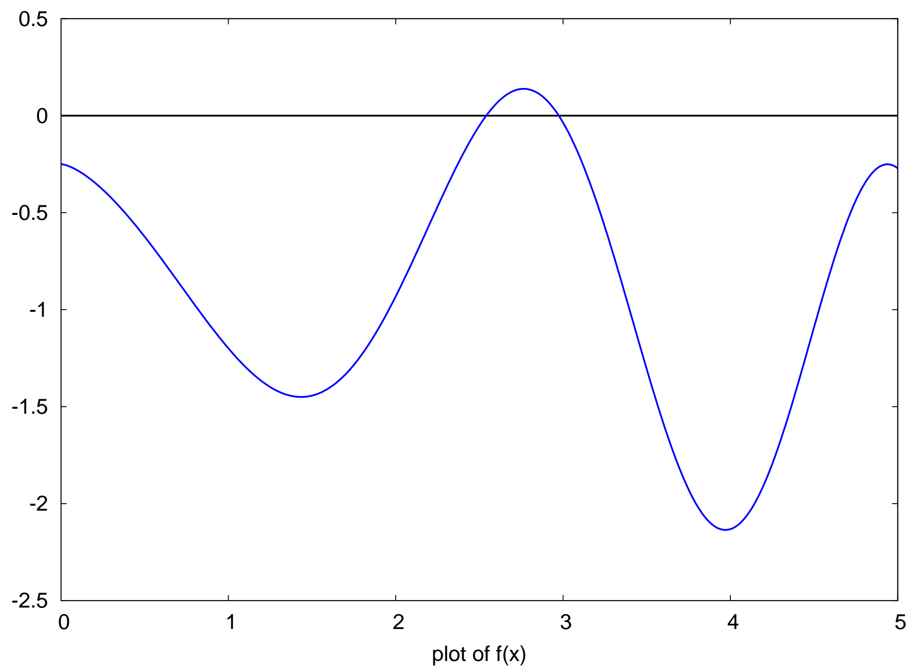

Figure 4: Plot of f(x)

This plot shows roots near the points `[x = 2.53, x = 2.97]`.

Here is our session making a plot and using **find_root**:

```
(%i1) fpprintprec:8$
(%i2) f(x):= cos(x/%pi)*exp(-(x/4)^2) - sin(x^(3/2)) - 5/4$

(%i3) plot2d(  f(x),[x,0,5],
              [style,  [lines,4,1] ],
               [xlabel," plot of f(x) "],[ylabel," "],
            [gnuplot_preamble, "set nokey; set xzeroaxis lw 2 "] )$
(%i4) [find_root(f,2.5,2.6), find_root(f, x, 2.5, 2.6),
        find_root(f(x),x,2.5,2.6), find_root(f(y),y,2.5,2.6)];
(%o4)           [2.5410501, 2.5410501, 2.5410501, 2.5410501]
(%i5) [x1 : find_root(f,2.5,2.6),x2 : find_root(f, 2.9, 3.0 )];
(%o5)                       [2.5410501, 2.9746034]
(%i6) float( map(f, [x1,x2] ) );
(%o6)                  [3.33066907E-16, 2.77555756E-16]
```

We see that the numerical accuracy of **find_root** using default behavior is the normal accuracy of Maxima arithmetic.

4.2.4 find_root: Quote that Function!

The Maxima function **find_root** is an unusual function. The source code which governs the behavior of **find_root** has been purposely designed to allow uses like the following:

```
(%i1) fpprintprec:8$
(%i2) find_root(diff(cos(2*x)*sin(3*x)/(1+x^2),x), x, 0, 0.5);
(%o2)                             0.321455
```

find_root evaluates the derivative and parks the resulting expression in the code function "fr(x)",say, to be used to probe where the function changes sign, as the code executes a bisection search for a root in the range called for. The code first checks that the sign of the function fr(x) has opposite signs at the end points of the given range. Next the code makes small steps in one direction, checking at each step if a sign change has occurred. As soon as a sign change has occurred, the code backs up one step, cuts the size of the step in half, say, and starts stepping again, looking for that sign change again. This is a brute force method which will find that root if there is one in the given interval. Of course the user can always evaluate the derivative and submit the resulting expression to **find_root** to find the same answer, as in:

```
(%i3) ex : trigsimp( diff(cos(2*x)*sin(3*x)/(1+x^2),x) );
             2
(%o3) - (((2 x  + 2) sin(2 x) + 2 x cos(2 x)) sin(3 x)
                                       2                          4      2
                          + (- 3 x  - 3) cos(2 x) cos(3 x))/(x  + 2 x  + 1)
(%i4) plot2d([0.0,ex],[x,-3,3])$
(%i5) find_root(ex,x,0,0.5);
(%o5)                             0.321455
```

If we assign the delayed derivative to an expression ex1, we can then use an argument `ev(ex1,diff)` to **find_root** as in:

```
(%i6) ex1 :  'diff(cos(2*x)*sin(3*x)/(1+x^2),x);
                              d   cos(2 x) sin(3 x)
(%o6)                         -- (-----------------)
                              dx        2
                                       x  + 1
(%i7) find_root(ev(ex1,diff),x,0,0.5);
(%o7)                                0.321455
```

If we assign the delayed derivative to a function g(x), and assign `ev(g(x),diff)` to another function k(x), we can use **find_root** as in:

```
(%i8) g(x) := 'diff(cos(2*x)*sin(3*x)/(1+x^2),x);
                                  d   cos(2 x) sin(3 x)
(%o8)                     g(x) := -- (-----------------)
                                  dx             2
                                            1 + x
(%i9) k(x) := ev(g(x),diff);
(%o9)                         k(x) := ev(g(x), diff)
(%i10) find_root(k(x),x,0,0.5);
(%o10)                               0.321455
```

or just use the syntax:

```
(%i11) find_root( ev(g(x),diff),x,0,0.5 );
(%o11)                               0.321455
```

However, the following syntax which leaves out the variable name produces an error message:

```
(%i12) find_root( k, 0, 0.5 );
Non-variable 2nd argument to diff:
0.0
#0: g(x=0.0)
#1: k(x=0.0)
 -- an error.  To debug this try debugmode(true);
```

In the above cases we want **find_root** to use the default initial evaluation of the first slot argument before going to work looking for the root.

The important thing to stress is that the **find_root** source code, by default, is designed to evaluate the first slot expression before beginning the checking of the sign of the resulting function (after evaluation) at the end points and proceeding with the bisection search. If the user wants to use (for some reason) a function f(x) in the first slot of **find_root** and wishes to prevent the initial evaluation of the first slot expression, the user should use the syntax `'(f(x))` for the first slot entry, rather than `f(x)`. However, the code is loosely written so that most of the time you can get correct results without using the single quote syntax `'(f(x))`. All of the following examples give the correct answer for this simple function.

```
(%i13) f(x) := x - cos(x)$
(%i14) [find_root( f, 0, 1), find_root( 'f, 0, 1),
          find_root( '(f), 0, 1), find_root( f(x), x, 0, 1),
             find_root( '( f(x)), 'x, 0, 1),find_root( '( f(x)), x, 0, 1),
               find_root( 'f(x), x, 0, 1)];
(%o14) [0.739085, 0.739085, 0.739085, 0.739085, 0.739085, 0.739085, 0.739085]
```

However, one can arrive at specialized homemade functions which require the strict syntax (a quoted function entry in the first slot) to behave correctly. Suppose we need to find the numerical roots of a function defined by an integral. The following is a toy model which uses such a function in a case where we know the answer ahead of time. Instead of directly looking for the roots of the function $f(x) = (x^2 - 5)$, we look for the roots of the function $\int_{\sqrt{5}}^{x} 2\,y\,d\,y$.

```
(%i1) fpprintprec : 8$
(%i2) ex : integrate(2*'y,'y,sqrt(5),x);
                                              2
                                             x     5
(%o2)                                     2 (-- - -)
                                             2     2
(%i3) ex : expand(ex);
                                             2
(%o3)                                       x  - 5
(%i4) define( f(x), ex );
                                                 2
(%o4)                                  f(x) := x  - 5
(%i5) solve( ex );
(%o5)                        [x = - sqrt(5), x = sqrt(5)]
(%i6) rr : float( map(rhs,%) );
(%o6)                           [- 2.236068, 2.236068]
(%i7) map(f,rr);
(%o7)                      [8.8817842E-16, 8.8817842E-16]
(%i8) find_root(f,0,4);
(%o8)                                   2.236068
```

Let's concentrate on finding the root `2.23606797749979` using the indirect route. Define a function $g(x)$ in terms of **integrate**:

```
(%i9) g(x) := block([numer,keepfloat,y],
          numer:true,keepfloat:true,
          integrate(2*y,y,sqrt(5),x) )$
(%i10) map(g, [1,2,3]);
(%o10)                           [- 4.0, - 1.0, 4.0]
(%i11) map(f, [1,2,3]);
(%o11)                              [- 4, - 1, 4]
(%i12) [find_root( g, 1, 4), find_root( g(x),x,1,4),
         find_root( '(g(x)),'x,1,4 ), find_root( 'g(x),x,1,4 )];
(%o12)                [2.236068, 2.236068, 2.236068, 2.236068]
```

We see that we have no problems with getting the function $g(x)$ to "work" with **find_root**. However, if we replace **integrate** with **quad_qags**, we find problems. First let's show the numerical integration routine **quad_qags** at work by itself:

```
(%i13) quad_qags(2*'y,'y,sqrt(5),2);
(%o13)                     [- 1.0, 1.11076513E-14, 21, 0]
(%i14) g(2.0);
(%o14)                                  - 1.0
```

The **quad_qags** function returns a list of four items, the first being the numerical value of the integral, the second being an estimate of the error of the answer calculated, the third being the number of function evaluations required, and the last an error code. The returned error code `0` means no problems were encountered, and we will write a function which ignores the error code returned, although in "real life" we would always want to report an error code value which was not `0`.

Here we define h(x) which employs the function **quad_qags** to carry out the numerical itegration.

```
(%i15) h(x) := block([numer,keepfloat,y,qlist],
          numer:true,keepfloat:true,
          qlist : quad_qags(2*y,y,sqrt(5),x),
           qlist[1] )$
(%i16) map(h,[1,2,3]);
(%o16)                         [- 4.0, - 1.0, 4.0]
(%i17) map(g,[1,2,3]);
(%o17)                         [- 4.0, - 1.0, 4.0]
```

The function $h(x)$ does the same job as $g(x)$, but uses **quad_qags** instead of **integrate**. Now let's use $h(x)$ in the Maxima function **find_root**.

```
(%i18) find_root( h(x),x,1,4);
function has same sign at endpoints
[f(1.0) = - 5.0, f(4.0) = - 5.0]
 -- an error.  To debug this try debugmode(true);
```

We see that the syntax `find_root( h(x),x,1,4)` produced an error due to the way **find_root** evaluates the first slot. Somehow **find_root** assigned `-5.0` to the internal function fr(x) used to look for the root, and in the first steps of that root location, checking for a difference in sign of fr(x) and the range end points, found the value `-5.0` at both ends. In effect, the code was working on the problem `find_root( -5.0, x, 1, 4)`.

The following methods succeed.

```
(%i19) [find_root( h, 1, 4),find_root( '(h(x)),'x,1,4 ),
        find_root( '(h(x)),x,1,4 ),find_root( 'h(x),x,1,4 )];
(%o19)               [2.236068, 2.236068, 2.236068, 2.236068]
```

4.2.5 newton

The Maxima function **newton** can also be used for numerical solutions of a single equation. The Maxima manual describes the syntax as:

> Function: **newton** `(expr, x, x_0, eps)`
> Returns an approximate solution of `expr = 0` by Newton's method, considering `expr` to be a function of one variable, `x`. The search begins with `x = x_0` and proceeds until `abs(expr) < eps` (with `expr` evaluated at the current value of `x`).
> `newton` allows undefined variables to appear in `expr`, so long as the termination test `abs(expr) < eps` evaluates to `true` or `false`. Thus it is not necessary that `expr` evaluate to a number.
> `load(newton1)` loads this function.

The two examples in the manual are instructive:

```
(%i1) fpprintprec:8$
(%i2) load (newton1);
(%o2) C:/PROGRA~1/MAXIMA~3.0/share/maxima/5.14.0/share/numeric/newton1.mac
(%i3) newton (cos (u), u, 1, 1/100);
(%o3)                              1.5706753
(%i4) ev (cos (u), u = %);
(%o4)                            1.21049633E-4
(%i5) assume (a > 0);
(%o5)                                [a > 0]
```

```
(%i6) newton (x^2 - a^2, x, a/2, a^2/100);
(%o6)                          1.0003049 a
(%i7) ev (x^2 - a^2, x = %);
                                            2
(%o7)                       6.09849048E-4 a
```

Of course both of these examples are found exactly by **solve**:

```
(%i8) solve( cos(x) );
`solve' is using arc-trig functions to get a solution.
Some solutions will be lost.
                                         %pi
(%o8)                              [x = ---]
                                          2
(%i9) float(%);
(%o9)                           [x = 1.5707963]
(%i10) solve( x^2 - a^2,x );
(%o10)                         [x = - a, x = a]
```

I find the source code (Windows XP) in the folder
`c:\Program Files\Maxima-5.14.0\share\maxima\5.14.0\share\numeric`. Here is the code in the file newton1.mac:

```
newton(exp,var,x0,eps):=
    block([xn,s,numer],
      numer:true,
      s:diff(exp,var),
      xn:x0,
      loop,
        if abs(subst(xn,var,exp))<eps then return(xn),
        xn:xn-subst(xn,var,exp)/subst(xn,var,s),
      go(loop)
    )$
```

We see that the code implements the Newton-Raphson algorithm. Given a function $f(x)$ and an initial guess x_g which can be assigned to, say, x^i, a closer approximation to the value of x for which $f(x) = 0$ is generated by

$$x^{i+1} = x^i - \frac{f(x^i)}{f'(x^i)}.$$

The method depends on being able to evaluate the derivative of $f(x)$ which appears in the denominator.
Steven Koonin's (edited) comments (Computational Physics: Fortran Version, Steven E. Koonin and Dawn Meredith, WestView Press, 1998, Ch. 1, Sec.3) are cautionary:

> When the function $f(x)$ is badly behaved near its root (e.g., there is an inflection point near the root) or when there are several roots, the "automatic" Newton-Raphson method can fail to converge at all or converge to the wrong answer if the initial guess for the root is poor.

4.3 Two or More Equations: Symbolic and Numerical Solutions

For sets of equations, we can use **solve** with the syntax:

> Function: **solve** `([eqn_1, ..., eqn_n], [x_1, ..., x_n])`
> `solve ([eqn_1, ..., eqn_n], [x_1, ..., x_n])` solves a system of simultaneous (linear or non-linear) polynomial equations by calling `linsolve` or `algsys` and returns a list of the solution lists in the variables. In the case of `linsolve` this list would contain a single list of solutions. This form of `solve` takes two lists as arguments. The first list represents the equations to be solved; the second list is a list of the unknowns to be determined. If the total number of variables in the equations is equal to the number of equations, the second argument-list may be omitted.

4.3.1 Numerical or Symbolic Linear Equations with solve or linsolve

The Maxima functions **solve**, **linsolve**, and **linsolve_by_lu** can be used for linear equations.

Linear equations containing symbolic coefficients can be "solved" by **solve** and **linsolve**. For example the pair of equations

$$ax + by = c, dx + ey = f$$

. Here we solve for the values of `(x,y)` which simultaneously satisfy these two equations and check the solutions.

```
(%i1) eqns : [a*x + b*y = c, d*x + e*y = f];
(%o1)                  [b y + a x = c, e y + d x = f]
(%i2) solve(eqns,[x,y]);
                             c e - b f      c d - a f
(%o2)                 [[x = - ---------, y = ---------]]
                             b d - a e      b d - a e
(%i3) soln : linsolve(eqns,[x,y]);
                             c e - b f      c d - a f
(%o3)                  [x = - ---------, y = ---------]
                             b d - a e      b d - a e
(%i4) (ev(eqns, soln), ratexpand(%%) );
(%o4)                          [c = c, f = f]
```

Note the presence of the determinant of the "coefficient matrix" in the denominator of the solutions.

A simple numerical (rather than symbolic) two equation example:

```
(%i1) eqns : [3*x-y=4,x+y=2];
(%o1)                     [3 x - y = 4, y + x = 2]
(%i2) solns : solve(eqns,[x,y]);
                                    3      1
(%o2)                        [[x = -, y = -]]
                                    2      2
(%i3) soln : solns[1];
                                    3      1
(%o3)                         [x = -, y = -]
                                    2      2
(%i4) for i thru 2 do disp( ev( eqns[i],soln ) )$
                                    4 = 4
                                    2 = 2
```

Using **linsolve** instead returns a list, rather than a list of a list.

```
(%i5) linsolve(eqns,[x,y]);
                                        3       1
(%o5)                              [x = -, y = -]
                                        2       2
```

4.3.2 Matrix Methods for Linear Equation Sets: linsolve_by_lu

The present version (5.14) of the Maxima manual does not have an index entry for the function **linsolve_by_lu**. These notes include only the simplest of many interesting examples described in two mailing list responses by the creator of the linear algebra package, Barton Willis (Dept. of Mathematics, Univ. Nebraska at Kearney), dated Oct. 27, 2007 and Nov. 21, 2007.

If we re-cast the two equation problem we have just been solving in the form of a matrix equation `A . xcol = bcol`, we need to construct the square matrix `A` so that matrix multiplication by the column vector `xcol` results in a column vector whose rows contain the left hand sides of the equations. The column vector `bcol` rows hold the right hand sides of the equations. Our notation below (as `xycol` and `xylist`) is only natural for a two equation problem (ie., a 2 x 2 matrix): you can invent your own notation to suit your problem.

If the argument of the function **matrix** is a simple list, the result is a row vector (a special case of a matrix object). We can then take the **transpose** of the row matrix to obtain a column matrix, such as `xcol` below. A direct definition of a two element column matrix is `matrix([x],[y])`, which is probably faster than `transpose(matrix([x,y]))`. To reduce the amount of space taken up by the default matrix output, we can set `display2d:false`.

```
(%i6) m : matrix( [3,-1],[1,1] );
                                    [ 3   - 1 ]
(%o6)                               [         ]
                                    [ 1    1  ]
(%i7) display2d:false$
(%i8) m;
(%o8) matrix([3,-1],[1,1])
(%i9) xcol : matrix([x],[y]);
(%o9) matrix([x],[y])
(%i10) m . xcol;
(%o10) matrix([3*x-y],[y+x])
```

The period allows non-commutative multiplication of matrices. The linear algebra package function **linsolve_by_lu** allows us to specify the column vector `bcol` as a simple list:

```
(%i11) b : [4,2];
(%o11) [4,2]
(%i12) linsolve_by_lu(m,b);
(%o12) [matrix([3/2],[1/2]),false]
(%i13) xycol : first(%);
(%o13) matrix([3/2],[1/2])
(%i14) m . xycol - b;
(%o14) matrix([0],[0])
(%i15) xylist : makelist( xycol[i,1],i,1,length(xycol) );
(%o15) [3/2,1/2]
(%i16) xyrules : map("=",[x,y],xylist);
(%o16) [x = 3/2,y = 1/2]
```

The **matrix** argument needs to be a square matrix. The output of **linsolve_by_lu** is a list: the first element is the solution column vector which for this two dimensional problem we have called `xycol`.

We check the solution in input `%i14`. The output `%o14` is a column vector each of whose elements is zero; such a column vector is ordinarily replaced by the number `0`.

One should always check solutions when using computer algebra software, since the are occasional bugs in the algorithms used. The second list element returned by **linsolve_by_lu** is **false**, which should always be the value returned when the calculation uses non-floating point numbers as we have here. If floating point numbers are used, the second element should be either **false** or a lower bound to the "matrix condition number". (We show an example later.) We have shown how to convert the returned `xycol` matrix object into an ordinary list, and how to then construct a list of replacement rules (as would be returned by **linsolve**) which could then be used for other purposes. The use of **makelist** is the recommended way to use parts of **matrix** objects in lists. However, here is a simple method which avoids **makelist**:

```
(%i17) flatten( args( xycol));
(%o17) [3/2,1/2]
```

but **makelist** should normally be the weapon of choice, since the method is foolproof and can be extended to many exotic ways of using the various elements of a matrix.

The Maxima function **linsolve_by_lu** allows the second argument to be either a list (as in the example above) or a column matrix, as we show here.

```
(%i18) bcol : matrix([4],[2])$
(%i19) linsolve_by_lu(m,bcol);
(%o19) [matrix([3/2],[1/2]),false]
(%i20) m . first(%) - bcol;
(%o20) matrix([0],[0])
```

4.3.3 Symbolic Linear Equation Solutions: Matrix Methods

Here we use **linsolve_by_lu** for the pair of equations

$$ax + by = c, \qquad dx + ey = f,$$

seeking the values of `(x,y)` which simultaneously satisfy these two equations and checking the solutions.

```
(%i1) display2d:false$
(%i2) m : matrix( [a,b], [d,e] )$
(%i3) bcol : matrix( [c], [f] )$
(%i4) ls : linsolve_by_lu(m,bcol);
(%o4) [matrix([(c-b*(f-c*d/a)/(e-b*d/a))/a],[(f-c*d/a)/(e-b*d/a)]),false]
(%i5) xycol : ratsimp( first(ls) );
(%o5) matrix([-(b*f-c*e)/(a*e-b*d)],[(a*f-c*d)/(a*e-b*d)])
(%i6) ( m . xycol - bcol, ratsimp(%%) );
(%o6) matrix([0],[0])
(%i7) (display2d:true,xycol);
                                          [    b f - c e ]
                                          [ - ---------- ]
                                          [    a e - b d ]
(%o7)                                     [              ]
                                          [   a f - c d  ]
                                          [   ---------  ]
                                          [   a e - b d  ]
(%i8) determinant(m);
(%o8)                                         a e - b d
```

We see the value of the determinant of the "coefficient matrix" m in the denominator of the solution.

4.3.4 Multiple Solutions from Multiple Right Hand Sides

Next we show how one can solve for multiple solutions (with one call to **linsolve_by_lu**) corresponding to a number of different "right hand sides". We will turn back on the default matrix display, and re-define the first (right hand side) column vector as `b1col`, and the corresponding solution `x1col`.

```
(%i21) display2d:true$
(%i22) b1col : matrix( [4], [2] );
                                   [ 4 ]
(%o22)                             [   ]
                                   [ 2 ]
(%i23) x1col : first( linsolve_by_lu(m,b1col) );
                                   [ 3 ]
                                   [ - ]
                                   [ 2 ]
(%o23)                             [   ]
                                   [ 1 ]
                                   [ - ]
                                   [ 2 ]
(%i24) b2col : matrix( [3], [1] );
                                   [ 3 ]
(%o24)                             [   ]
                                   [ 1 ]
(%i25) x2col : first( linsolve_by_lu(m, b2col) );
                                   [ 1 ]
(%o25)                             [   ]
                                   [ 0 ]
(%i26) bmat : matrix( [4,3], [2,1] );
                                 [ 4  3 ]
(%o26)                           [      ]
                                 [ 2  1 ]
(%i27) linsolve_by_lu( m, bmat );
                              [ 3     ]
                              [ -   1 ]
                              [ 2     ]
(%o27)                       [[       ], false]
                              [ 1     ]
                              [ -   0 ]
                              [ 2     ]
(%i28) xsolns : first(%);
                                 [ 3     ]
                                 [ -   1 ]
                                 [ 2     ]
(%o28)                           [       ]
                                 [ 1     ]
                                 [ -   0 ]
                                 [ 2     ]
```

```
(%i29) m . xsolns - bmat;
                                   [ 0  0 ]
(%o29)                             [      ]
                                   [ 0  0 ]
(%i30) x1col : col(xsolns,1);
                                     [ 3 ]
                                     [ - ]
                                     [ 2 ]
(%o30)                               [   ]
                                     [ 1 ]
                                     [ - ]
                                     [ 2 ]
(%i31) x2col : col(xsolns,2);
                                     [ 1 ]
(%o31)                               [   ]
                                     [ 0 ]
```

In input `%i26` we define the `2 x 2` matrix `bmat` whose first column is `b1col` and whose second column is `b2col`. Using `bmat` as the second argument to **linsolve_by_lu** results in a return list whose first element (what we call `xsolns`) is a `2 x 2` matrix whose first column is `x1col` (the solution vector corresponding to `b1col`) and whose second column is `x2col` (the solution vector corresonding to `b2col`). In input `%i29` we check both solutions simultaneously. The result is a matrix with every element equal to zero, which would ordinarily be replaced by the number `0`. In input `%i30` we extract `x1col` using the **col** function.

4.3.5 Three Linear Equation Example

We next consider a simple three linear equation example. Although **solve** does not require the list `[x,y,z]` in this problem, if you leave it out, the solution list returned will be in an order determined by the peculiarities of the code, rather than by you. By including the variable list as `[x,y,z]`, you are forcing the output list to be in the same order.

```
(%i1) eqns : [2*x - 3*y + 4*z = 2, 3*x - 2*y + z = 0,
                     x + y - z = 1]$
(%i2) display2d:false$
(%i3) solns : solve( eqns,[x,y,z] );
(%o3) [[x = 7/10,y = 9/5,z = 3/2]]
(%i4) soln : solns[1];
(%o4) [x = 7/10,y = 9/5,z = 3/2]
(%i5) for i thru 3 do disp( ev(eqns[i],soln) )$
2 = 2
0 = 0
1 = 1
```

The Maxima function **linsolve** has the same syntax as **solve** (for a set of equations) but you cannot leave out the list of unknowns.

```
(%i6) linsolve(eqns,[x,y,z]);
(%o6) [x = 7/10,y = 9/5,z = 3/2]
```

Notice that **linsolve** returns a list, while **solve** returns a list of lists.

Next we use **linsolve_by_lu** on this three equation problem. Using the laws of matrix multiplication, we "reverse engineer" the 3×3 matrix `m` and the three element column vector `bcol` which provide an equivalent problem in matrix form.

```
(%i7) m : matrix( [2,-3,4],[3,-2,1],[1,1,-1] )$
(%i8) xcol : matrix( [x],[y],[z] )$
(%i9) m . xcol;
(%o9) matrix([4*z-3*y+2*x],[z-2*y+3*x],[-z+y+x])
(%i10) bcol : matrix( [2],[0],[1] )$
(%i11) linsolve_by_lu(m,bcol);
(%o11) [matrix([7/10],[9/5],[3/2]),false]
(%i12) xyzcol : first(%);
(%o12) matrix([7/10],[9/5],[3/2])
(%i13) m . xyzcol - bcol;
(%o13) matrix([0],[0],[0])
(%i14) xyzlist : makelist( xyzcol[i,1],i,1,length(xyzcol) );
(%o14) [7/10,9/5,3/2]
(%i15) xyzrules : map("=",[x,y,z],xyzlist);
(%o15) [x = 7/10,y = 9/5,z = 3/2]
```

Both **linsolve** and **solve** can handle an "equation list" which is actually an "expression list" in which it is understood that the required equations are generated by setting each expression to zero.

```
(%i16) exs : [2*x - 3*y + 4*z - 2, 3*x - 2*y + z ,
                    x + y - z - 1];
(%o16)       [4 z - 3 y + 2 x - 2, z - 2 y + 3 x, - z + y + x - 1]
(%i17) linsolve(exs,[x,y,z]);
                                    7       9      3
(%o17)                         [x = --, y = -, z = -]
                                    10      5      2
(%i18) solve(exs);
                                    3      9      7
(%o18)                        [[z = -, y = -, x = --]]
                                    2      5      10
```

The Maxima manual presents a linear equation example in which there is an undefined parameter "a".

```
(%i1) eqns : [x + z = y,2*a*x - y = 2*a^2,y - 2*z = 2]$
(%i2) solns : linsolve(eqns,[x,y,z] );
(%o2)                   [x = a + 1, y = 2 a, z = a - 1]
(%i3) solve(eqns,[x,y,z]);
(%o3)                  [[x = a + 1, y = 2 a, z = a - 1]]
(%i4) for i thru 3 do (
          e: expand( ev(eqns[i],solns) ),disp(lhs(e) - rhs(e)) )$
                                       0
                                       0
                                       0
(%i5) e;
(%o5)                                2 = 2
(%i6) [kill(e),e];
(%o6)                              [done, e]
```

The code in input `%i4` binds an equation to the symbol `e` which can be removed with **kill**.

We can avoid introducing a global binding to a symbol by using `%%`, which allows use of the previous result inside a piece of code.

```
(%i7) for i thru 3 do (
            expand( ev(eqns[i],solns) ),disp(lhs(%%) - rhs(%%)) )$
                                      0
                                      0
                                      0
```

Note the syntax used here: `do ( job1, job2 ) .`

Let's try out **linsolve_by_lu** on this three linear (in the unknowns `(x,y,z)`) equation problem which involves the unbound parameter `a`.

```
(%i8) display2d:false$
(%i9) m : matrix([1,-1,1],[2*a,-1,0],[0,1,-2] )$
(%i10) xcol : matrix( [x],[y],[z] )$
(%i11) m . xcol;
(%o11) matrix([z-y+x],[2*a*x-y],[y-2*z])
(%i12) bcol : matrix( [0],[2*a^2], [2] )$
(%i13) soln : linsolve_by_lu(m,bcol)$
(%i14) xyzcol : ( first(soln), ratsimp(%%) );
(%o14) matrix([a+1],[2*a],[a-1])
(%i15) ratsimp( m . xyzcol - bcol);
(%o15) matrix([0],[0],[0])
```

In input `%i15` we check the solution, and the result is a three element column vector, all of whose elements are zero. Such a column matrix is ordinarily replaced by the number `0`.

4.3.6 Surpressing rat Messages: ratprint

If we start with two linear equations with decimal coefficients, **solve** (and the other methods) converts the decimals to ratios of integers, and tries to find an exact solution. You can avoid seeing all the rational replacements done by setting the option variable **ratprint** to **false**. Despite the assertion, in the manual section on **rat**, that

> `keepfloat` if `true` prevents floating point numbers from being converted to rational numbers.

setting **keepfloat** to **true** here does not stop **solve** from converting decimal numbers to ratios of integers.

```
(%i1) [keepfloat,ratprint];
(%o1)                            [false, true]
(%i2) display2d:false$
(%i3) fpprintprec:8$
(%i4) eqns : [0.2*x + 0.3*y = 3.3,0.1*x - 0.8*y = 6.6]$
(%i5) solns : solve(eqns, [x,y]);
`rat' replaced -3.3 by -33/10 = -3.3
`rat' replaced 0.2 by 1/5 = 0.2
`rat' replaced 0.3 by 3/10 = 0.3
`rat' replaced -6.6 by -33/5 = -6.6
`rat' replaced 0.1 by 1/10 = 0.1
`rat' replaced -0.8 by -4/5 = -0.8
(%o5) [[x = 462/19,y = -99/19]]
(%i6) linsolve(eqns,[x,y]);
`rat' replaced -3.3 by -33/10 = -3.3
```

```
'rat' replaced 0.2 by 1/5 = 0.2
'rat' replaced 0.3 by 3/10 = 0.3
'rat' replaced -6.6 by -33/5 = -6.6
'rat' replaced 0.1 by 1/10 = 0.1
'rat' replaced -0.8 by -4/5 = -0.8
(%o6) [x = 462/19,y = -99/19]
(%i7) m : matrix([0.2,0.3],[0.1,-0.8] )$
(%i8) bcol : matrix( [3.3], [6.6] )$
(%i9) linsolve_by_lu(m,bcol);
'rat' replaced 0.2 by 1/5 = 0.2
'rat' replaced 0.2 by 1/5 = 0.2
'rat' replaced 0.2 by 1/5 = 0.2
'rat' replaced 0.95 by 19/20 = 0.95
(%o9) [matrix([24.315789],[-5.2105263]),false]
(%i10) ratprint:false$
(%i11) solns : solve(eqns, [x,y]);
(%o11) [[x = 462/19,y = -99/19]]
(%i12) linsolve(eqns,[x,y]);
(%o12) [x = 462/19,y = -99/19]
(%i13) linsolve_by_lu(m,bcol);
(%o13) [matrix([24.315789],[-5.2105263]),false]
(%i14) float(solns);
(%o14) [[x = 24.315789,y = -5.2105263]]
```

Matrix methods for sets of linear equations can be solved using IEEE double floats (as well as "big floats") by including an optional "method" specification `'floatfield` after the input column vector (or matrix of input column vectors).

```
(%i15) linsolve_by_lu(m,bcol, 'floatfield);
(%o15) [matrix([24.315789],[-5.2105263]),8.8815789]
```

In this example the lower bound of the matrix condition number appears as the second element of the returned list.

4.3.7 Non-Linear Polynomial Equations

Here is an example of using **solve** to find a pair of exact solutions of a pair of equations, one equation being linear, the other quadratic. The pair of solutions represent the two intersections of the unit circle with the line $y = -x/3$.

```
(%i1) fpprintprec:8$
(%i2) eqns : [x^2 + y^2 = 1, x + 3*y = 0]$
(%i3) solns : solve(eqns,[x,y]);
                  3                 1
(%o3) [[x = - --------, y = ---------------],
              sqrt(10)      sqrt(2) sqrt(5)
                                         3                  1
                                [x = --------, y = - ---------------]]
                                     sqrt(10)        sqrt(2) sqrt(5)
```

```
(%i4) solns : rootscontract(solns);
                    3            1               3              1
(%o4)  [[x = - --------, y = --------], [x = --------, y = - --------]]
               sqrt(10)      sqrt(10)        sqrt(10)        sqrt(10)
(%i5) for i thru 2 do for j thru 2 do (
                ev(eqns[i],solns[j]), disp(lhs(%%)-rhs(%%)) )$
                                      0
                                      0
                                      0
                                      0
(%i6) float(solns);
(%o6)  [[x = - 0.948683, y = 0.316228], [x = 0.948683, y = - 0.316228]]
```

The pair of solutions reflect the symmetry of the given set of equations, which remain invariant under the transformation $x \rightarrow -y, y \rightarrow -x$.

A set of two nonlinear polynomial equations with four solutions generated by **solve** is one of the examples in the Maxima manual. One of the solutions is exact, one is a real inexact solution, and the other two solutions are inexact complex solutions.

```
(%i1)  fpprintprec:8$
(%i2)  eqns : [4*x^2 - y^2 = 12, x*y - x = 2]$
(%i3)  solns : solve( eqns, [x,y] );
(%o3) [[x = 2, y = 2], [x = 0.520259 %i - 0.133124,
y = 0.0767838 - 3.6080032 %i], [x = - 0.520259 %i - 0.133124,
y = 3.6080032 %i + 0.0767838], [x = - 1.7337518, y = - 0.153568]]
(%i4) for i thru 2 do for j thru length(solns) do (
      expand( ev(eqns[i],solns[j]) ),
      abs( lhs(%%) - rhs(%%) ), disp(%%) )$
                                      0
                                2.36036653E-15
                                2.36036653E-15
                                 1.13954405E-6
                                      0
                                     0.0
                                     0.0
                                 9.38499825E-8
```

To get real numbers from the complex solutions, we used the **abs** function, which calculates the absolute value of a complex number. Note the syntax used to check the solutions: `do ( job1, job2, job3).`

4.3.8 General Sets of Nonlinear Equations: eliminate, mnewton

Solving systems of nonlinear equations is much more difficult than solving one nonlinear equation. A wider variety of behavior is possible: determining the existence and number of solutions or even a good starting guess is more complicated. There is no method which can guarantee convergence to the desired solution. The computing labor increases rapidly with the number of dimensions of the problem.

4.3.9 Intersections of Two Circles: implicit_plot

Given two circles, we seek the intersections points. We first write down the defining equations of the two circles, and look visually for points `(x,y)` which simultaneously lie on each circle. We use **implicit_plot** for this visual search.

```
(%i1) [eq1 : x^2 + y^2 = 1,eq2 : (x-2)^2 + (y-2)^2 = 4]$
(%i2) eqns : [eq1,eq2]$
(%i3) load(implicit_plot);
(%o3)
    C:/PROGRA~1/MAXIMA~3.0/share/maxima/5.14.0/share/contrib/implicit_plot.lisp
(%i4) implicit_plot(eqns,[x,-6,6],[y,-6,6],[nticks,200],
          [gnuplot_preamble, "set zeroaxis" ])$
```

We are not taking enough care with the `x` and `y` ranges to make the "circles" circular, but we can use the cursor to read off approximate intersections points: `(x,y) = (0.26,0.98), (0.96,0.30)`. However, the defining equations are invariant under the symmetry transformation $x \leftrightarrow y$, so the solution pairs must also respect this symmetry. We next eliminate `y` between the two equations and use **solve** to find accurate values for the `x`'s. Since we know that both solutions have positive values for `y`, we enforce this condition on equation 1.

```
(%i5) solve(eq1,y);
                                          2                    2
(%o5)                [y = - sqrt(1 - x ), y = sqrt(1 - x )]
(%i6) ysoln : second(%);
                                                  2
(%o6)                            y = sqrt(1 - x )
(%i7) eliminate(eqns,[y]);
                                       2
(%o7)                           [32 x  - 40 x + 9]
(%i8) xex : solve(first(%));
                               sqrt(7) - 5      sqrt(7) + 5
(%o8)                 [x = - -----------, x = -----------]
                                    8                8
(%i9) (fpprintprec:8, xex : float(xex) );
(%o9)                    [x = 0.294281, x = 0.955719]
(%i10) [x1soln : first(xex), x2soln : second(xex) ]$
(%i11) [ev(%o7,x1soln), ev(%o7,x2soln)];
(%o11)                   [[- 4.4408921E-16], [0.0]]
(%i12) y1soln : ev(ysoln,x1soln);
(%o12)                          y = 0.955719
(%i13) y2soln : ev(ysoln,x2soln);
(%o13)                          y = 0.294281
(%i14) [soln1:[x1soln,y1soln],soln2:[x2soln,y2soln] ]$
(%i15) [ev(eqns,soln1), ev(eqns,soln2) ];
(%o15)             [[1.0 = 1, 4.0 = 4], [1.0 = 1, 4.0 = 4]]
(%i16) [soln1,soln2];
(%o16)   [[x = 0.294281, y = 0.955719], [x = 0.955719, y = 0.294281]]
```

We have solutions (%o16) which respect the symmetry of the equations. The solutions have been numerically checked in input %i15.

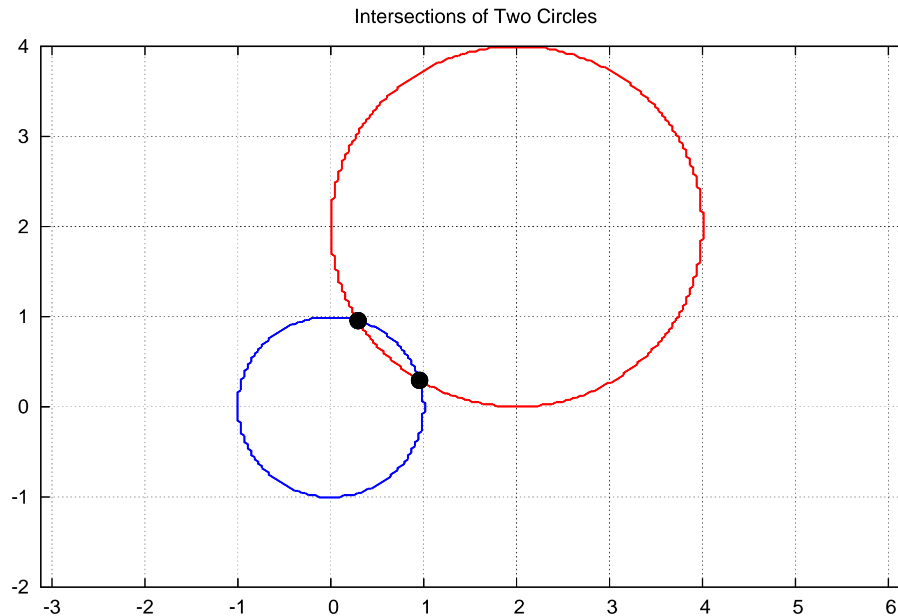

Figure 5: two circles

4.3.10 Using Draw for Implicit Plots

The figure above was created using the draw package with the following code in a separate work file "implicitplot1.mac". The code the the file is

```
/* file implicitplot1.mac  */
/*  need load(implicit_plot);  to use this code  */
disp(" doplot() ")$

doplot() := block([ x,y, eq1, eq2, y1:-2, y2:4,r, x1 ,x2 ],
                  r : 1.56,
                  x1 : r*y1,
                  x2 : r*y2,
                  eq1 : x^2 + y^2 = 1,
              eq2 : (x-2)^2 + (y-2)^2 = 4,
          draw2d(
             grid      = true,
             line_type = solid,
             line_width  = 3,
             color = blue,
             implicit(eq1, x, x1,x2, y, y1,y2),
             color = red,
             implicit(eq2, x, x1,x2, y, y1,y2),
             color = black,
             point_type = filled_circle,
             point_size = 2,
```

```
            points( [ [0.294281,0.955719], [ 0.955719, 0.294281] ] ),
            title     = "Intersections of Two Circles" ,
            terminal = 'eps ,
  file_name = "c:/work2/mycircle2" )
        )$
```

Here is record of use:

```
(%i1) load(draw);
(%o1)   C:/PROGRA~1/MAXIMA~3.0/share/maxima/5.14.0/share/draw/draw.lisp
(%i2) load(implicitplot1);
                                     doplot()

(%o2)                     c:/work2/implicitplot1.mac
(%i3) doplot();
(%o3)                 [gr2d(implicit, implicit, points)]
```

Note that no actual plot was drawn on the screen, since an ”eps” file was created ”mycircle2.eps” in my work folder `c:\work2` for use in my latex file. To use this code to get a figure on your screen, you would remove the last two lines: `terminal = 'eps , file_name = "c:/work2/mycircle2"` and, very important!, also remove the comma at the end of ”two circles”.

4.3.11 Another Example

We next work through Example 6.5.1 (page 149), in ”Numerical Methods for Unconstrained Optimization and Nonlinear Equations” by J. E. Dennis, Jr. and Robert B. Schnabel (Prentice-Hall, 1983).

```
(%i1) eq1 : x^2 + y^2 = 2;
                                     2    2
(%o1)                               y  + x  = 2
(%i2) eq2 : exp(x - 1) + y^3 = 2;
                                  3     x - 1
(%o2)                            y  + %e      = 2
```

We will concentrate on the solution `(x,y) = (1,1)`. We can eliminate `y` and plot the resulting function of `x` to visually locate the `x` solutions.

```
(%i3) eliminate([eq1,eq2],[y]);
           2 x       x + 1     2  6       2  4        2  2       2
(%o3)    [%e    - 4 %e      + %e  x  - 6 %e  x  + 12 %e  x  - 4 %e ]
(%i4) ex : first(%);
           2 x       x + 1     2  6       2  4        2  2       2
(%o4)     %e    - 4 %e      + %e  x  - 6 %e  x  + 12 %e  x  - 4 %e
(%i5) plot2d([0.0,ex],[x,-5,5] )$
(%i6) plot2d([0.0,ex],[x,0,2] )$
(%i7) x1 : find_root(ex,x,0.5,1.5);
(%o7)                                 1.0
(%i8) yeqn : ev(eq1,x = x1);
                                    2
(%o8)                              y  + 1.0 = 2
(%i9) solve(yeqn);
`rat' replaced -1.0 by -1/1 = -1.0
(%o9)                            [y = - 1, y = 1]
```

```
(%i10) ysol : second(%);
(%o10)                              y = 1
(%i11) soln1 : [x = x1, ysol];
(%o11)                        [x = 1.0, y = 1]
(%i12) ev(eq1,soln1);
(%o12)                             2.0 = 2
(%i13) ev(eq2,soln1);
(%o13)                             2.0 = 2
```

One solution is then `%o11` and the solution has been checked.

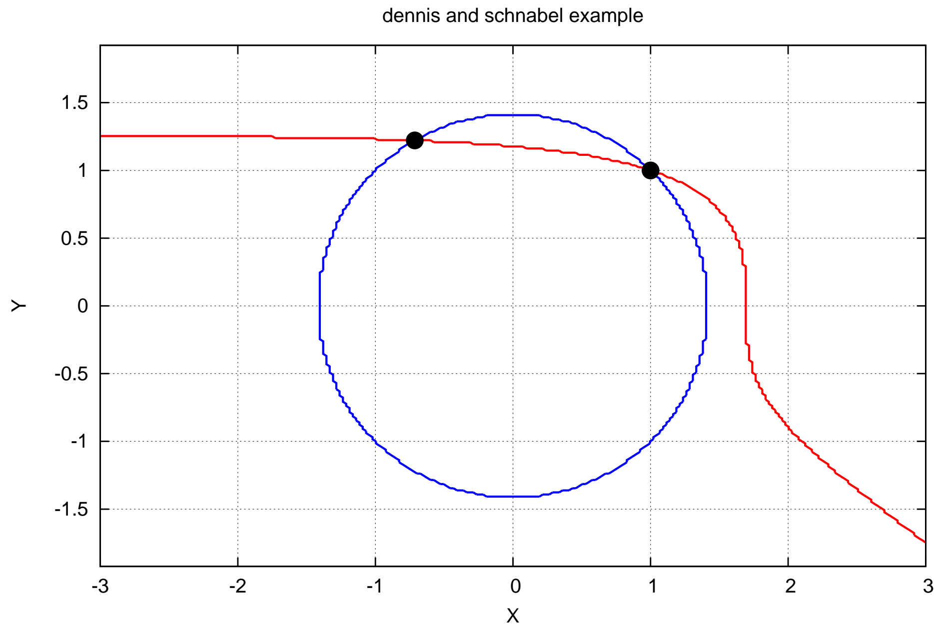

Figure 6: Dennis and Schnabel Example

Let's try **mnewton** on this problem. Wikipedia, under "Newton's Method", has the introduction:

> In numerical analysis, Newton's method (also known as the Newton Raphson method or the Newton Fourier method) is perhaps the best known method for finding successively better approximations to the zeros (or roots) of a real valued function. Newton's method can often converge remarkably quickly, especially if the iteration begins "sufficiently near" the desired root. Just how near "sufficiently near" needs to be and just how quickly "remarkably quickly" can be depends on the problem, as is discussed in detail below. Unfortunately, far from the desired root, Newton's method can easily lead an unwary user astray, and astray with little warning. Such users are advised to heed the advice of Press, et. al. (1992), who suggest embedding Newton's method in a routine that also detects possible convergence failures.
> Newton's method can also be used to find a minimum or maximum of such a function, by finding a zero in the function's first derivative, see Newton's method as an optimization algorithm.
> The algorithm is first in the class of Householder's methods, succeeded by Halley's method.

The Maxima manual entry for **mnewton** begins:

> Function: **mnewton** `(FuncList,VarList,GuessList)`
> Multiple nonlinear functions solution using the Newton method. FuncList is the list of functions to solve, VarList is the list of variable names, and GuessList is the list of initial approximations.
> The solution is returned in the same format that solve() returns. If the solution isn't found, `[]` is returned.
> This function is controlled by global variables `newtonepsilon` and `newtonmaxiter`.

Here are the Maxima manual examples of **mnewton**:

```
(%i1) load(mnewton);
(%o1) C:/PROGRA~1/MAXIMA~3.0/share/maxima/5.14.0/share/contrib/mnewton.mac
(%i2) mnewton([2*a^a-5],[a],[1]);

(%o2)                       [[a = 1.70927556786144]]
(%i3) mnewton([2*3^u-v/u-5, u+2^v-4], [u, v], [2, 2]);
(%o3)            [[u = 1.066618389595407, v = 1.552564766841786]]
(%i4) mnewton([x1+3*log(x1)-x2^2, 2*x1^2-x1*x2-5*x1+1],
              [x1, x2], [5, 5]);
(%o4)         [[x1 = 3.756834008012769, x2 = 2.779849592817898]]
```

In the above examples, **mnewton** is presented with a list of expressions. Here we use **mnewton** on the Dennis and Schnabel problem we solved earlier using eliminate and find_root. We rewrite the equations as expressions here.

```
(%i1) fpprintprec:8$
(%i2) load(mnewton);
(%o2) C:/PROGRA~1/MAXIMA~3.0/share/maxima/5.14.0/share/contrib/mnewton.mac
(%i3) exs : [x^2 + y^2 -2, exp(x-1)+y^3-2]$
(%i4) mn(x0,y0) := mnewton(exs,[x,y],[x0,y0] )$
(%i5) mn(1.1,0.9);
(%o5)                         [[x = 1.0, y = 1.0]]
(%i6) mn(1.2,0.8);
(%o6)                         [[x = 1.0, y = 1.0]]
(%i7) mn(1.3,0.7);
(%o7)                         [[x = 1.0, y = 1.0]]
(%i8) mn(1.4,0.6);
(%o8)                         [[x = 1.0, y = 1.0]]
(%i9) mn(1.5,0.5);
(%o9)                   [[x = - 0.713747, y = 1.2208868]]
(%i10) mn(1.5,0.6);
(%o10)                        [[x = 1.0, y = 1.0]]
(%i11) mn(1.7,0.6);
(%o11)                        [[x = 1.0, y = 1.0]]
(%i12) mn(1.9,0.6);
(%o12)                  [[x = - 0.713747, y = 1.2208868]]
(%i13) mn(1.9,0.7);
(%o13)                        [[x = 1.0, y = 1.0]]
(%i14) mn(2,0.7);
(%o14)                        [[x = 1.0, y = 1.0]]
(%i15) mn(0.8,1.1);
(%o15)                        [[x = 1.0, y = 1.0]]
(%i16) mn(0.5,1.2);
(%o16)                        [[x = 1.0, y = 1.0]]
(%i17) mn(0.1,1.2);
(%o17)                  [[x = - 0.713747, y = 1.2208868]]
```

We have to be "close enough" to find the (1,1) root with **mnewton**. Note that Maxima's function **mnewton** can't find the desired root starting with `(x0,y0) = (2,0.5)` as the textbook example does with eventual success.

4.3.12 Error Messages and Do It Yourself Mnewton

Let's explore the Newton Raphson method using an easy example which **solve** has no trouble with. As a by-product, we show that **mnewton** can deal with a list of equations, rather than a list of expressions. After getting an error message from Maxima's **mnewton**, we work this same problem "by hand", using matrix methods. A good reference is Chapter 5, Nonlinear Equations, of the text "Scientific Computing: An Introductory Survey" (2nd ed.), by Michael T. Heath. (see webpage:
`http://www.cse.uiuc.edu/heath/scicomp/pubdata/index.html`)

```
(%i18) eqns : [x+y=3,x^2+y^2=9]$
(%i19) mn(x0,y0) := mnewton(eqns,[x,y],[x0,y0] )$
(%i20) solve(eqns,[x,y]);
(%o20)                [[x = 3, y = 0], [x = 0, y = 3]]
(%i21) mn(1,2);
(%o21)                 [[x = 6.37714165E-17, y = 3.0]]
(%i22) mn(-1,4);
(%o22)                 [[x = 6.37714165E-17, y = 3.0]]
(%i23) mn(-2,5);
(%o23)                 [[x = 1.13978659E-16, y = 3.0]]
(%i24) mn(0,0);
Maxima encountered a Lisp error:

 Error in FUNCALL [or a callee]: Zero divisor.

Automatically continuing.
To reenable the Lisp debugger set *debugger-hook* to nil.
(%i25) mn(0,1);
(%o25)                 [[x = 1.13978659E-16, y = 3.0]]
(%i26) mn(2,0);
(%o26)                 [[x = 3.0, y = 1.41267055E-16]]
```

The "zero divisor" message from the Maxima code for **mnewton** probably means that the starting point `(x,y) = (0,0)` resulted in an attempted division by 0. To explore what kind of problems can arise, we implement a naive (ie., a strategy-less) iteration algorithm as presented in Heath's text. We present the algorithm first in terms of a 2×2 matrix and a two element column vector which are functions of the scalars `(x,y)`. We then convert the algorithm to a form which works with two element column vectors `v` and `b`. We let the column vector `b` hold the elements `x` and `y` and we seek `b` such that the equation $f(b) = 0$, where `f` is a column vector which defines the problem. If `b` is an approximate solution of this equation, then $bbetter = b + s$, where `s` is the solution of the matrix equation `j(b) . s = -f(b)`, and the 2×2 matrix `j(b)` is the jacobian matrix: `j[1,1] : diff( f[1,1],x )`, `j[1,2] : diff( f[1,1],y )`, and `j[2,1] : diff( f[2,1],x)`, `j[2,2] : diff( f[2,1],y )`.

```
(%i1) (fpprintprec:8,ratprint:false)$
(%i2) g : matrix( [x + y -3],[x^2 + y^2 -9] );
                                 [   y + x - 3  ]
(%o2)                            [              ]
                                 [   2    2     ]
                                 [ y  + x  - 9 ]
```

```
(%i3) gv : ev(g, x=v[1,1], y=v[2,1] );
                              [ v     + v     - 3 ]
                              [  2, 1    1, 1     ]
(%o3)                         [                   ]
                              [  2       2        ]
                              [ v     + v     - 9 ]
                              [  2, 1    1, 1     ]
(%i4) define(f(v), gv);
                                  [ v     + v     - 3 ]
                                  [  2, 1    1, 1     ]
(%o4)                     f(v) := [                   ]
                                  [  2       2        ]
                                  [ v     + v     - 9 ]
                                  [  2, 1    1, 1     ]
(%i5) b : matrix([1],[2]);
                                      [ 1 ]
(%o5)                                 [   ]
                                      [ 2 ]
(%i6) f(b);
                                     [  0  ]
(%o6)                                [     ]
                                     [ - 4 ]
(%i7) (r1 : g[1,1], r2 : g[2,1] )$
(%i8) h : matrix( [diff(r1,x), diff(r1,y)],
              [diff(r2,x), diff(r2,y) ] );
                                  [  1    1  ]
(%o8)                             [          ]
                                  [ 2 x  2 y ]
(%i9) hv : ev(h, x=v[1,1], y=v[2,1] );
                              [    1        1    ]
(%o9)                         [                  ]
                              [ 2 v      2 v     ]
                              [    1, 1     2, 1 ]
(%i10) define( j(v), hv );
                                  [    1        1    ]
(%o10)                    j(v) := [                  ]
                                  [ 2 v      2 v     ]
                                  [    1, 1     2, 1 ]
(%i11) j(b);
                                    [ 1  1 ]
(%o11)                              [      ]
                                    [ 2  4 ]
(%i12) ls : linsolve_by_lu(j(b),-f(b) );
                                 [ - 2 ]
(%o12)                          [[     ], false]
                                 [  2  ]
(%i13) s :  first(ls) ;
                                     [ - 2 ]
(%o13)                               [     ]
                                     [  2  ]
```

```
(%i14) b : b + s;
                                   [ - 1 ]
(%o14)                             [     ]
                                   [  4  ]
(%i15) b : b + first( linsolve_by_lu(j(b),-f(b) )  );
                                   [   1 ]
                                   [ - - ]
                                   [   5 ]
(%o15)                             [     ]
                                   [ 16  ]
                                   [ --  ]
                                   [ 5   ]
(%i16) b : b + first( linsolve_by_lu(j(b),-f(b) )  );
                                  [   1  ]
                                  [ - -- ]
                                  [   85 ]
(%o16)                            [      ]
                                  [ 256  ]
                                  [ ---  ]
                                  [ 85   ]
(%i17) b : b + first( linsolve_by_lu(j(b),-f(b) )  );
                                 [     1   ]
                                 [ - ----- ]
                                 [   21845 ]
(%o17)                           [         ]
                                 [  65536  ]
                                 [  -----  ]
                                 [  21845  ]
(%i18) b : b + float( first( linsolve_by_lu(j(b),-f(b) )  ) );
                            [ - 6.98491931E-10 ]
(%o18)                      [                  ]
                            [       3.0        ]
(%i19) f(b);
                              [      0.0      ]
(%o19)                        [               ]
                              [ 4.19095159E-9 ]
```

Starting with the guess `(x0,y0) = (1,2)`, this iteration process has converged to the approximate solution given by `%o18`, which we check as an approximate solution in input `%i19`. Now let's start with the "dangerous" guess: `(x0,y0) = (0,0)`.

```
(%i20) b : matrix( [0],  [0] );
                                    [ 0 ]
(%o20)                              [   ]
                                    [ 0 ]
(%i21) f(b);
                                   [ - 3 ]
(%o21)                             [     ]
                                   [ - 9 ]
(%i22) j(b);
                                  [ 1  1 ]
(%o22)                            [      ]
                                  [ 0  0 ]
```

```
(%i23) ls : linsolve_by_lu(j(b),-f(b) );
Division by 0
 -- an error.  To debug this try debugmode(true);
(%i24) determinant(j(b));
(%o24)                                0
```

Thus, a check of the non-vanishing of the determinant of the jacobian matrix would have kept us out of trouble.

4.3.13 Automated Code for mymnewton

Writing code for an arbitrary number of dimensions is a suggested homework problem. Here we just assume the problem is two dimensional and assume the variables are called `x` and `y`. To check for the ”nonvanishing” of the determinant of the jacobian, we ask if the absolute value is less than some very small number. Here is the code, written with notepad2 in a file ”mymnewton.mac”, placed in my work directory `c:\work2\`. You can download this file from the author’s webpage, and experiment with it. You can reduce the size of the output on the screen by adding the line ”`display2d:false,`” in the program, or outside the program in your work session. If you make changes to this code, add some extra ”debug” printouts at first like ”`display(newval1,newval2)`”, or ”`print(" v1 = ",v1),`” to make sure you are on the right track. Once the program has been ”debugged”, you can comment out the debug version in your work file, copy the whole code to a new section, remove the debug printouts, and use as your ”working version”.

```
/*  working version  */
/* file: mymnewton.mac
    e. woollett, april, 08  */


disp("working version mymnewton,
assumes two dimensional problem only,
syntax:

mymnewton( exprlist, guesslist, numiter )$

exprlist should have the form: [expr1, expr2],
to find (x,y) such that simultaneously expr1=0,expr2=0,
expr1, expr2 should be explicit functions of x and y,
guesslist should be in the form: [xguess,yguess],
numiter = number of iterations ")$

mymnewton( exprlist, guesslist, numiter) :=

   block([numer,ratprint,fpprintprec, small : 1.0e-30 ,
                g,x,y,gv,h,hv,b,v,ls,d  ] , local(f,j),

     numer:true, ratprint:false, fpprintprec:8,
      /* g = col vec: row 1=expr 1, row 2=expr 2 depends on (x,y) */

     g : matrix( [ exprlist[1] ],[ exprlist[2] ] ),
     display(g),

     gv : ev(g, x=v[1,1], y=v[2,1] ),

      /* v is generic col vector */
```

```
  define( f(v), gv ),

  /* h is jacobian matrix associated with col vec g(x,y)  */

  h : matrix( [diff( g[1,1], x), diff( g[1,1], y)],
        [diff( g[2,1], x), diff(g[2,1], y) ] ),

  hv : ev(h, x=v[1,1], y=v[2,1] ),

  define( j(v), hv ),

  /* b is col vec containing (x,y) values  */

  b : matrix([ guesslist[1] ],[ guesslist[2] ]),

      /*  start iterations  */

  for i:0 thru numiter do (

    print("  "),
    print(" i = ",i,"    b = ",b,"   f(b) = ",f(b)  ),
    if ( i = 0) then print(" starting values ") else
              print("    condition number = ", second(ls) ),

          /*  check jacobian determinant  */

    d : ( determinant( j(b) ), float(%%) , abs(%%) ),

    if ( d < small ) then (
           print("  abs(det(jacobian)) is ", d,"  program halt " ),
                             return()  ),

     /*  using 'floatfield arg gets condition number returned */

    ls : linsolve_by_lu(j(b),-f(b), 'floatfield ),

     /*  improved (hopefully) estimates of values of (x,y)
         which simultaneously satisfy expr1=0 and expr2=0 */

    b : b +  first(ls)

   )  /* end do  */

)$   /*  end block  */
```

and here is a little output from this code:

```
(%i1) load("mymnewton.mac");
working version mymnewton,
assumes two dimensional problem only,
syntax:
mymnewton( exprlist, guesslist, numiter )$
exprlist should have the form: [expr1, expr2],
to find (x,y) such that simultaneously expr1=0,expr2=0,
expr1, expr2 should be explicit functions of x and y,
guesslist should be in the form: [xguess,yguess],
numiter = number of iterations
(%o1)                      c:/work2/mymnewton.mac
(%i2) mymnewton([x+y-3, x^2+y^2-9],[1,2], 5)$
                                    [  y + x - 3  ]
                               g = [              ]
                                    [  2    2     ]
                                    [ y  + x  - 9 ]



                  [ 1 ]            [  0  ]
 i =  0     b =  [   ]    f(b) =  [     ]
                  [ 2 ]            [ - 4 ]
 starting values

                  [ - 1.0 ]            [ 0.0 ]
 i =  1     b =  [       ]    f(b) =  [     ]
                  [  4.0  ]            [ 8.0 ]
    condition number =  22.5

                  [ - 0.2 ]            [ 0.0  ]
 i =  2     b =  [       ]    f(b) =  [      ]
                  [  3.2  ]            [ 1.28 ]
    condition number =  19.5

                  [ - 0.0117647 ]            [    0.0    ]
 i =  3     b =  [             ]    f(b) =  [           ]
                  [  3.0117647  ]            [ 0.0708651 ]
    condition number =  10.92

                  [ - 4.57770657E-5 ]            [ 4.4408921E-16 ]
 i =  4     b =  [                 ]    f(b) =  [               ]
                  [    3.0000458    ]            [ 2.74666585E-4 ]
    condition number =  7.212872

                  [ - 6.98491837E-10 ]            [      0.0      ]
 i =  5     b =  [                  ]    f(b) =  [               ]
                  [       3.0        ]            [ 4.19095159E-9 ]
    condition number =  7.000824
```

Here is a test of the "dangerous" initial condition case:

```
(%i3) mymnewton([x+y-3, x^2+y^2-9],[0,0], 5)$
                                      [   y + x - 3  ]
                                  g = [              ]
                                      [   2    2     ]
                                      [ y  + x  - 9  ]


                    [ 0 ]             [ - 3 ]
 i =  0      b =  [   ]     f(b) =  [     ]
                    [ 0 ]             [ - 9 ]
 starting values
  abs(det(jacobian)) is  0.0   program halt
```

It is left as a homework problem to incorporate a criterion for finding an "accurate enough" solution without providing the program a number of iterations, and allowing the program to "hide" the details of the iteration process, providing the user with an "answer" and an estimate of the likely "error".

Maxima by Example: Ch.5: 2D Plots and Graphics using qdraw *

Edwin L. Woollett

January 29, 2009

Contents

5 2D Plots and Graphics using qdraw 3

5.1 Quick Plots for Explicit Functions: ex(...) . . . 3
5.2 Quick Plots for Implicit Functions: imp(...) . . . 10
5.3 Contour Plots with contour(...) . . . 12
5.4 Density Plots with qdensity(...) . . . 14
5.5 Explicit Plots with Greater Control: ex1(...) . . . 17
5.6 Explicit Plots with ex1(...) and Log Scaled Axes . . . 19
5.7 Data Plots with Error Bars: pts(...) and errorbars(...) . . . 21
5.8 Implicit Plots with Greater Control: imp1(...) . . . 27
5.9 Parametric Plots with para(...) . . . 29
5.10 Polar Plots with polar(...) . . . 31
5.11 Geometric Figures: line(...) . . . 32
5.12 Geometric Figures: rect(...) . . . 34
5.13 Geometric Figures: poly(...) . . . 35
5.14 Geometric Figures: circle(...) and ellipse(...) . . . 38
5.15 Geometric Figures: vector(..) . . . 40
5.16 Geometric Figures: arrowhead(..) . . . 43
5.17 Labels with Greek Letters . . . 43
- 5.17.1 Enhanced Postscript Methods . . . 43
- 5.17.2 Windows Fonts Methods with jpeg Files . . . 47
- 5.17.3 Using Windows Fonts with the Gnuplot Console Window . . . 48

5.18 Even More with more(...) . . . 49
5.19 Programming Homework Exercises . . . 50
- 5.19.1 General Comments . . . 50
- 5.19.2 XMaxima Tips . . . 51
- 5.19.3 Suggested Projects . . . 51

5.20 Acknowledgements . . . 52

*This version uses Maxima 5.17.1. This is a live document. Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions to `woollett@charter.net`

COPYING AND DISTRIBUTION POLICY

This document is part of a series of notes titled "Maxima by Example" and is made available via the author's webpage `http://www.csulb.edu/~woollett/` to aid new users of the Maxima computer algebra system.

NON-PROFIT PRINTING AND DISTRIBUTION IS PERMITTED.
You may make copies of this document and distribute them to others as long as you charge no more than the costs of printing.

These notes (with some modifications) will be published in book form eventually via Lulu.com in an arrangement which will continue to allow unlimited free download of the pdf files as well as the option of ordering a low cost paperbound version of these notes.

5 2D Plots and Graphics using qdraw

5.1 Quick Plots for Explicit Functions: ex(...)

This chapter provides an introduction to a new graphics interface developed by the author of the Maxima by Example tutorial notes. The **qdraw** package (qdraw.mac: available for download on the Maxima by Example webpage) is an interface to the **draw** package function **draw2d**; to obtain a plot you must load **draw** as well as **qdraw**. You can just use `load(qdraw)` if you have the file in your work folder and have set up your file search as described in Chap. 1. Otherwise just put qdraw.mac into your `...maxima...share\draw` folder where it will be found.

The primary motivation for the **qdraw** package is to provide "quick" (hence the "q" in "qdraw") plotting software which provides the kinds of plotting defaults which are of interest to students and researchers in the physical sciences and engineering. There are two "quick" plotting functions you can use with **qdraw**: **ex(...)** and **imp(...)**.

An entry like

```
(%i1) load(draw)$
(%i2) load(qdraw)$
(%i3)  qdraw( ex( [x,x^2,x^3],x,-3,3 ) )$
```

will produce a plot of the three explicit functions of `x` in the first argument list, using line width = 3, an automatic rotating series of default colors, clearly visible `x` and `y` axes, and also a "grid" as well. The **ex** function passes its arguments on to a series of calls to **draw2d**'s **explicit** function.

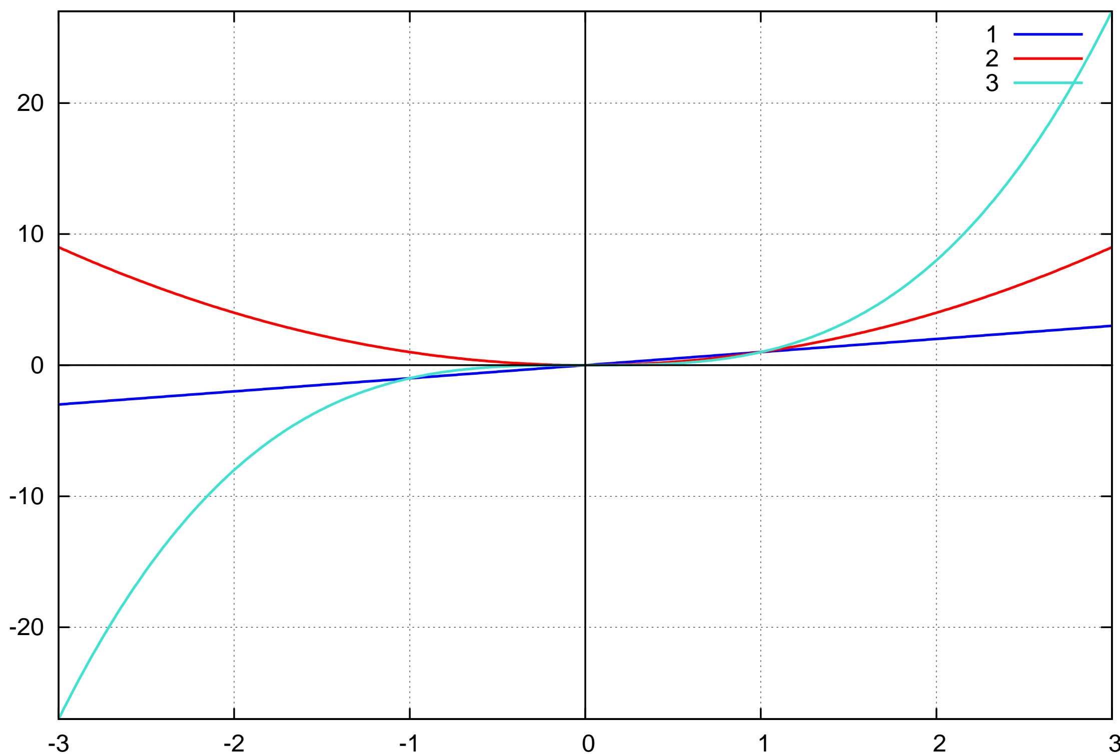

Figure 1: Using ex() for x, x^2, x^3

Since $3^3 = 27$, **draw2d** extends the vertical axis to $\pm\,27$ by default.

You can control the vertical range of the "canvas" with the **yr(...)** function, which passes its arguments to a **draw2d** entry `yrange = [y1,y2]`.

```
(%i4) qdraw( ex([x,x^2,x^3],x,-3,3 ),yr(-2,2) )$
```

which produces the plot:

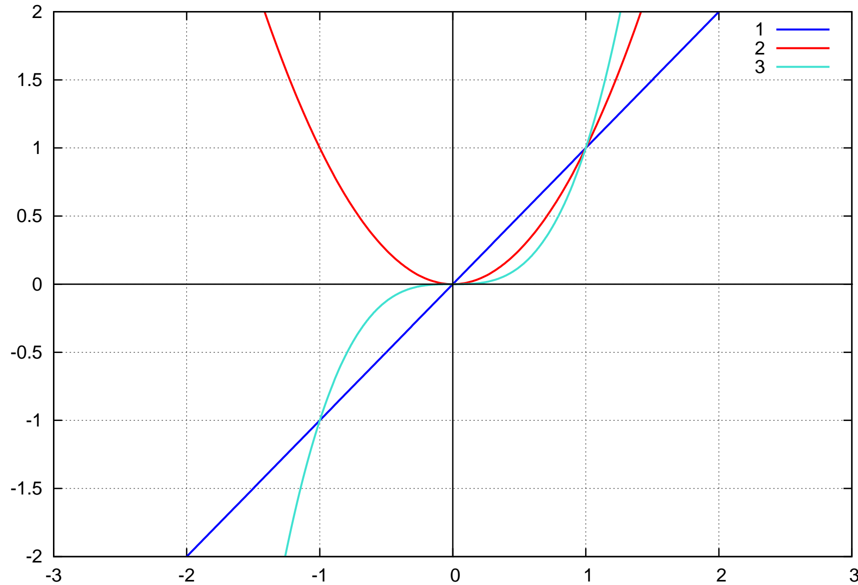

Figure 2: Adding $yr(-2,2)$

You can make the lines thinner or thicker than the default line width (3) by using the **lw(n)** option, which only affects the quick plotting functions **ex(...)** and **imp(...)**, as in

```
(%i5) qdraw(ex([x,x^2,x^3],x,-3,3 ),yr(-2,2), lw(6))$
```

to get:

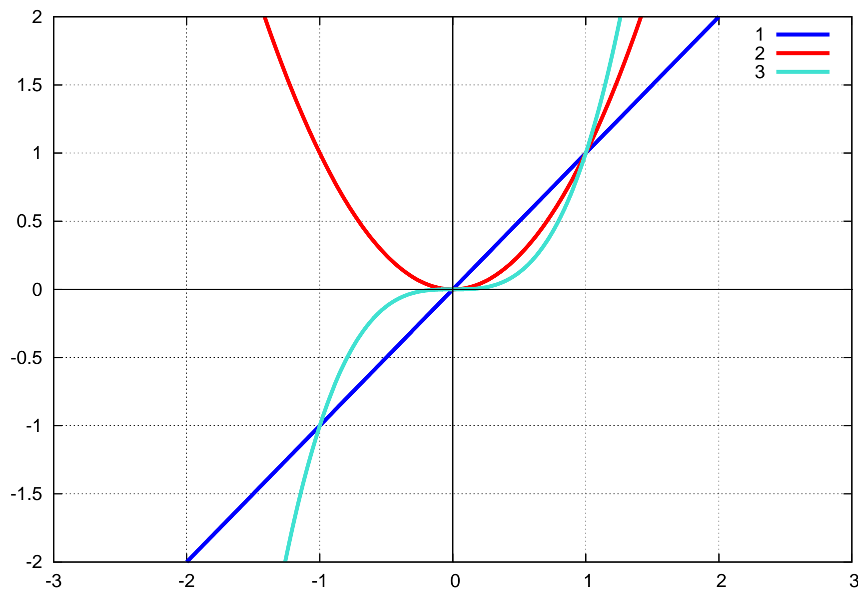

Figure 3: Adding $lw(6)$

You can place the plot "key" (legend) at the bottom right by using the **key(bottom)** option, as in:

```
(%i6) qdraw( ex( [x,x^2,x^3],x,-3,3 ),yr(-2,2),lw(6),key(bottom) )$
```

to get:

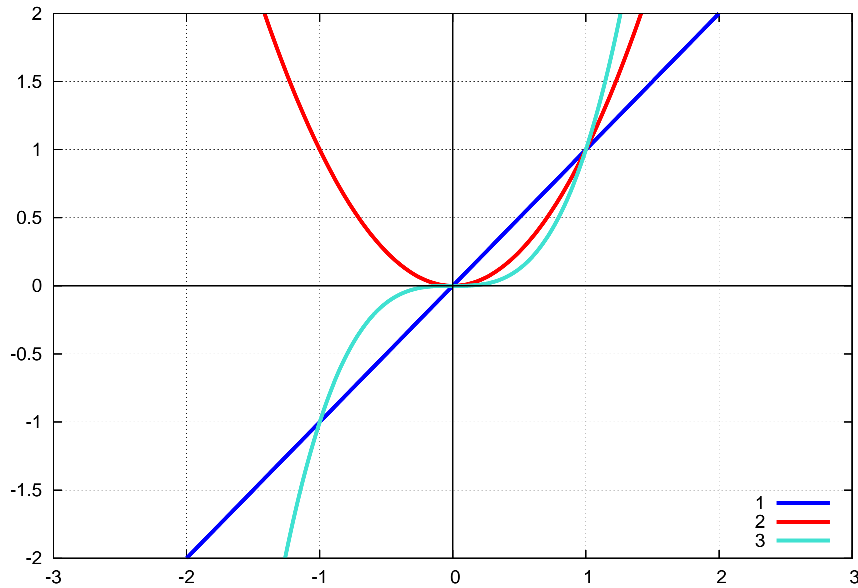

Figure 4: Adding $key(bottom)$

You can remove the default grid and xy axes by adding **cut(grid,xyaxes)** as in:

```
(%i7) qdraw( ex( [x,x^2,x^3],x,-3,3 ),yr(-2,2),
                lw(6),key(bottom), cut(grid, xyaxes) )$
```

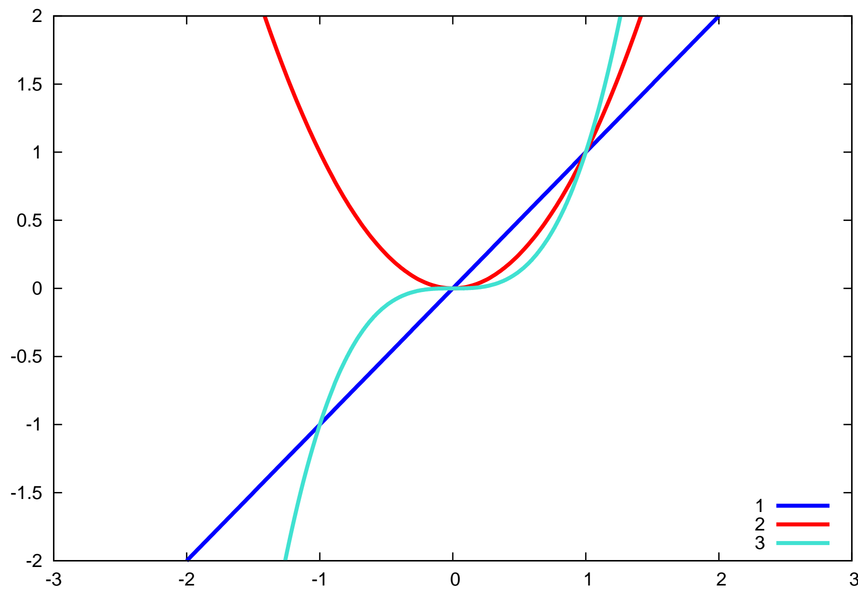

Figure 5: Adding $cut(grid, xyaxes)$

You can remove the grid, axes, the key, and all the borders using **cut(all)**, as in:

```
(%i8) qdraw( ex( [x,x^2,x^3],x,-3,3 ),yr(-2,2),
                 lw(6), cut( all ) )$
```

which results in a "clean" canvas:

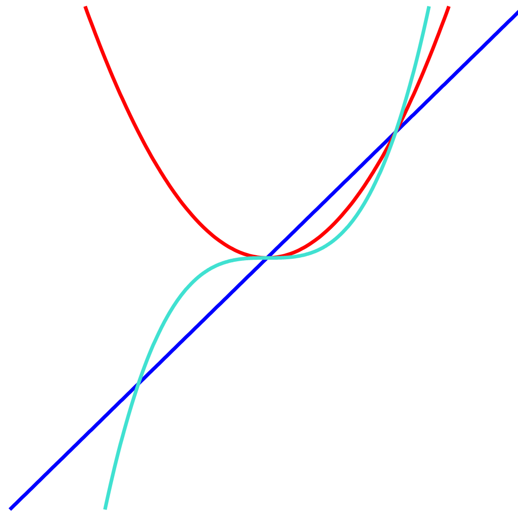

Figure 6: Adding $cut(all)$

Restoring the (default) grid and axes, we can place points (default size 3 and color black) at the intersection points using the **pts(...)** option, which passes a points list to **draw2d**'s **points** function:

```
(%i9)  qdraw( ex([x,x^2,x^3],x,-3,3 ),yr(-2,2),lw(6),
              key(bottom),pts( [ [-1,-1],[0,0],[1,1] ] ) )$
```

which produces:

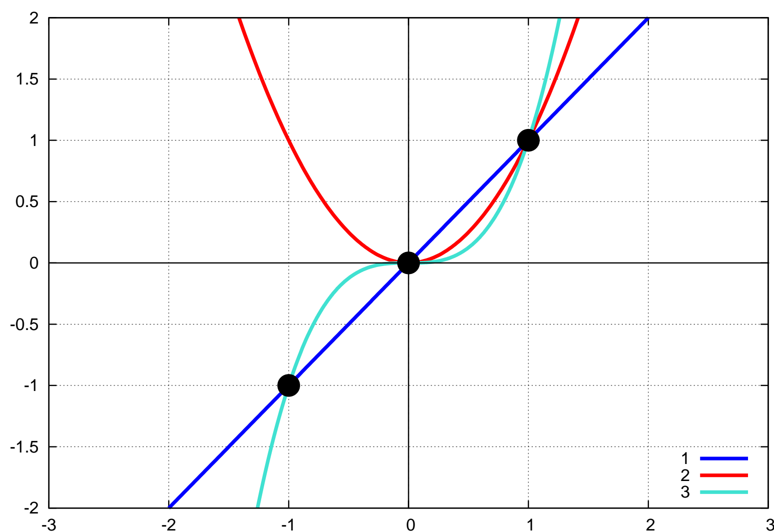

Figure 7: Adding $pts(ptlist)$

We can overide the default size and color of those points by including inside the **pts** function the optional **ps(n)** and **pc(c)** arguments, as in:

```
(%i10) qdraw( ex( [x,x^2,x^3],x,-3,3 ),yr(-2,2), lw(6),key(bottom),
              pts([ [-1,-1],[0,0],[1,1] ],ps(2),pc(magenta) ) )$
```

which produces:

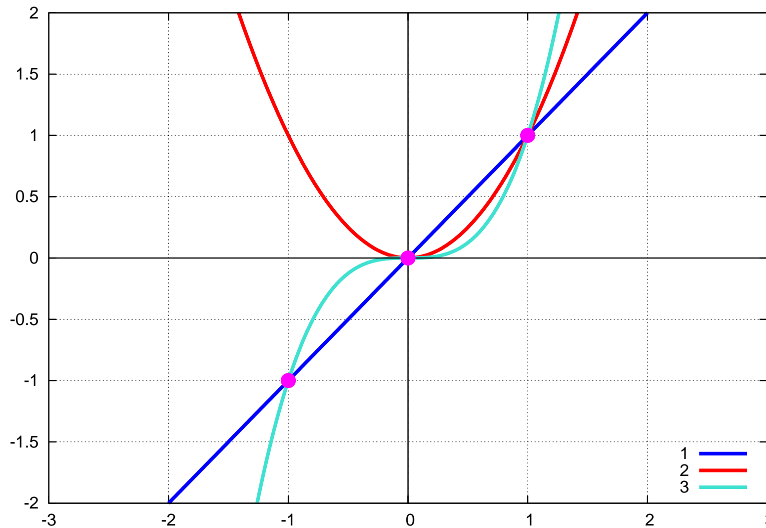

Figure 8: Adding $pts(ptlist, ps(2), pc(magenta))$

We can include a key entry for the points using the **pk(string)** option for the **pts** function, as in:

```
(%i11) qdraw( ex( [x,x^2,x^3],x,-3,3 ),yr(-2,2), lw(6),key(bottom),
          pts([ [-1,-1],[0,0],[1,1] ],ps(2),pc(magenta),pk("intersections") ) )$
```

which produces:

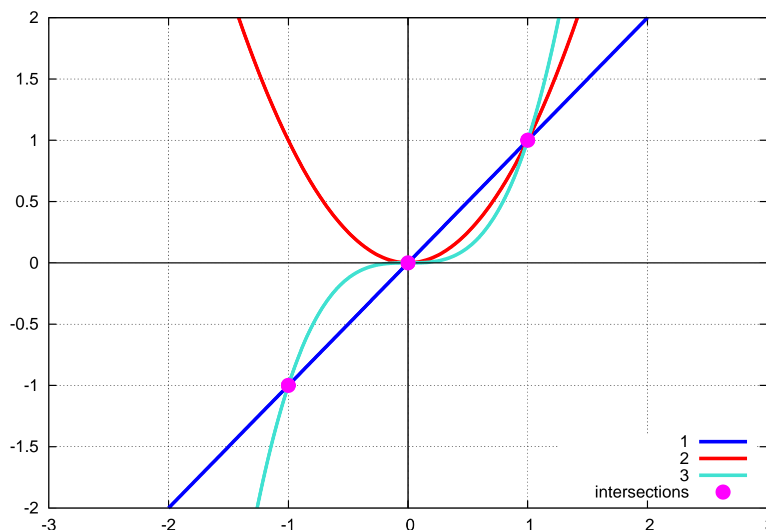

Figure 9: $pts(ptlist, ps(2), pc(magenta), pk("intersections"))$

The "eps" file ch5p9.eps used to get the last figure in the Tex file which is the source of this pdf file was produced using the **pic**(type, filename) option to **qdraw**, as in:

```
(%i12) qdraw( ex( [x,x^2,x^3],x,-3,3 ),yr(-2,2), lw(6),key(bottom),
        line(-3,0,3,0,lw(2)),line(0,-2,0,2,lw(2)),
    pts([ [-1,-1],[0,0],[1,1] ],ps(2),pc(magenta),pk("intersections")),
            pic(eps,"ch5p9") )$
```

We have discussed, at the end of Chapter 1, Getting Started, how we insert such an "eps" file into our Tex file in order to get the figures you see here.

The extra, optional, arguments we have included inside **qdraw** can be entered in any order; in fact, all arguments to **qdraw** are optional and can be entered in any order. For example

```
(%i13) qdraw( yr(-2,2),lw(6), ex( [x,x^2,x^3],x,-3,3 ),
              key(bottom), ex(sin(3*x)*exp(x/3),x,-3,3),
              pts([ [-1,-1],[0,0],[1,1] ]) )$
```

which adds $sin(3\,x)\,e^{x/3}$ with a separate **ex(...)** argument to **qdraw**, and produces

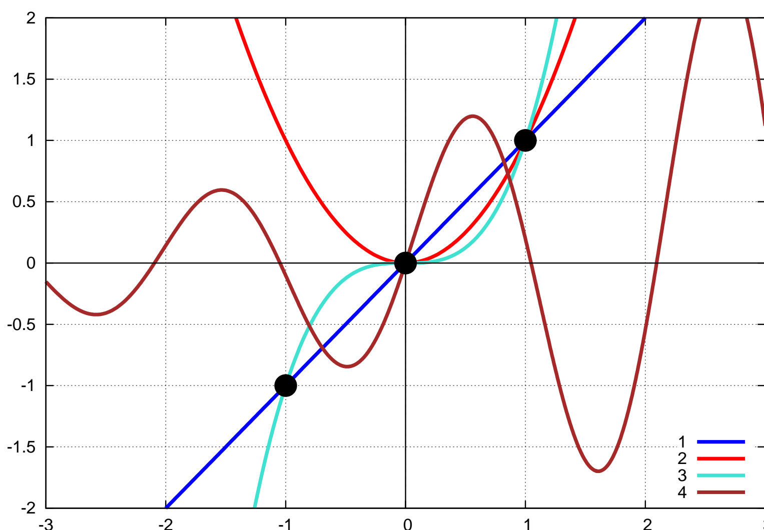

Figure 10: Adding $sin(3\,x)\,e^{x/3}$

We next add a label "qdraw at work" to our plot.

Using Windows, the font size must be adjusted only after getting the plot drawn in a Gnuplot window by right clicking the icon in the upper left hand corner, and selecting Options, Choose Font,... If you increase the windows graphics font from the default value of 10 to 20, say, you will see a dramatic increase in the size of the label, but also in the size of the x and y axis coordinate numbers, and also a large increase in size of any features of the graphics which used a call to draw2d's **points** function (such as **qdraw**'s **pts** function).

This behavior seems to be related to the limitations of the present incarnation of the adaptation of Gnuplot to the Windows system, and hopefully will be addressed in the future by the volunteers who work on Gnuplot software.

Our illustration of the use of labels will simply be what one gets by sending the graphics object to a graphics file "ch5p11.eps" and including that file in our Tex/pdf file. In Maxima, we use the code:

```
(%i14) qdraw( yr(-2,2),lw(6), ex( [x,x^2,x^3],x,-3,3 ),
              key(bottom), ex(sin(3*x)*exp(x/3),x,-3,3),
              pts([ [-1,-1],[0,0],[1,1] ]) ,
               label(["qdraw at work",-2.9,1.5]),
              pic(eps,"ch5p11",font("Times-Bold",20) ) );
```

The resulting plot is then

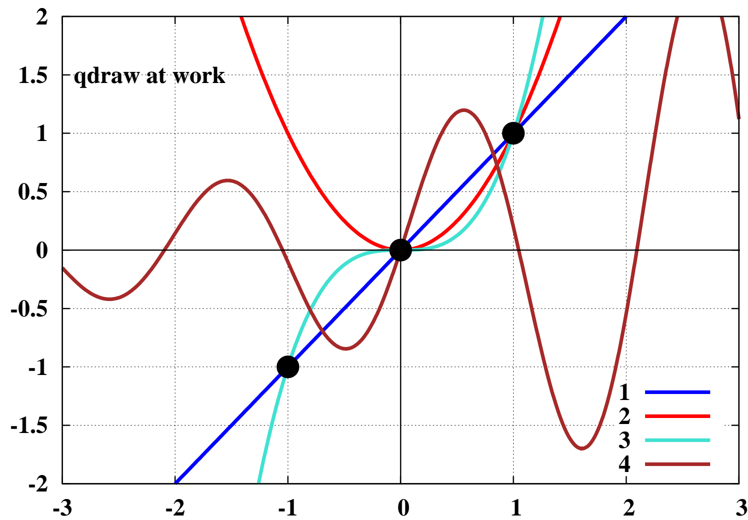

Figure 11: Adding a label at (-2.9,1.5)

If you look in the html Maxima manual under the index item "font", which takes you to a subsection of the draw package documentation, you will find a listing of the available postscript fonts, which one can use with the **pic(eps, filename, font(name, size))** function call. These options have the names Times-Roman, Times-Bold, Helvetica, Helvetica-Bold, Courier, Courier-Bold, also -Italic options.

If you want to save the graphics as a jpeg file, the font name should be a string containing the path to the desired font file. Using the Windows XP operating system, the available windows fonts are in the folder `c:\windows\fonts\`. Here is Maxima code to get a jpeg graphics file based on our present drawing:

```
(%i15) qdraw( yr(-2,2),lw(6), ex( [x,x^2,x^3],x,-3,3 ),
              key(bottom), ex(sin(3*x)*exp(x/3),x,-3,3),
              pts([ [-1,-1],[0,0],[1,1] ]) ,
               label(["qdraw at work",-2.9,1.5]),
              pic(jpg,"ch5p11",font("c:/windows/fonts/timesbd.ttf",20) ) );
```

The resulting jpeg file has thicker lines and bolder labels, so some experimentation may be called for to get the desired result. The font file requested corresponds to times roman bold. The font file extension "ttf" stands for "true type fonts". If you look in the windows, fonts folder you can find other interesting choices.

5.2 Quick Plots for Implicit Functions: imp(...)

The quick plotting function **imp(...)** has the syntax

```
      imp( eqnlist,  x,x1,x2,y,y1,y2 )
```

or

```
      imp( eqn,      x,x1,x2,y,y1,y2 ) .
```

If the equation(s) are actually functions of (u,v) then $x \rightarrow u$ and $y \rightarrow v$. The numbers (x1,x2) determine the horizontal canvas extent, and the numbers (y1,y2) determine the vertical canvas extent. Here is an example using the single equation form:

```
(%i16) qdraw( imp( sin(2*x)*cos(y)=0.4,x,-3,3,y,-3,3 ) ,
              cut(key) );
```

which produces the "implicit plot":

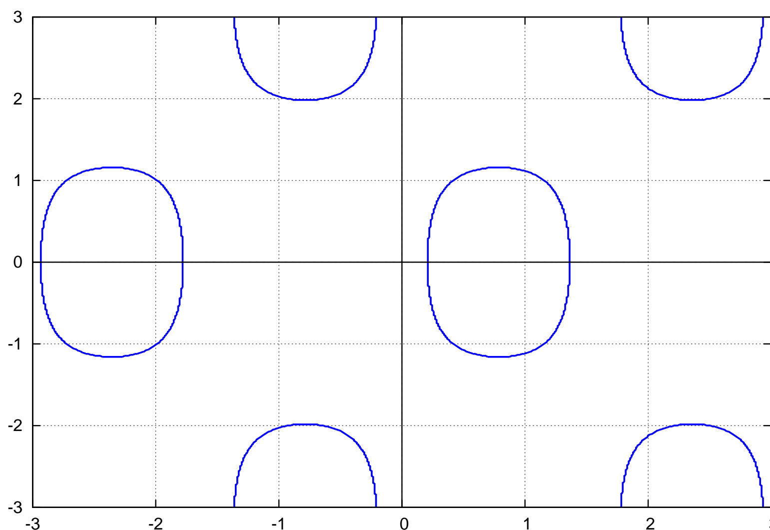

Figure 12: Implicit plot of $sin(2\,x)\,cos(y)$

which uses the default line width = 3, the first of the default rotating colors (blue), and, of course, the default axes and grid. To remove the default key, we have used the **cut** function. Since the left hand side of this equation will periodically return to the same numerical value in both the x and the y directions, there is no "limit" to the solutions obtained by setting the left hand side equal to some numerical value between zero and one.

This looks like one piece of a contour plot for the given function. We can add more contour lines using the **imp** function by using the list_of_equations form:

```
(%i17) qdraw( imp( [sin(2*x)*cos(y)=0.4,
              sin(2*x)*cos(y)=0.7,
              sin(2*x)*cos(y)=0.9] ,x,-3,3,y,-3,3 ) ,
              cut(key) );
```

The resulting plot with the default rotating color set is shown on the top of the next page.

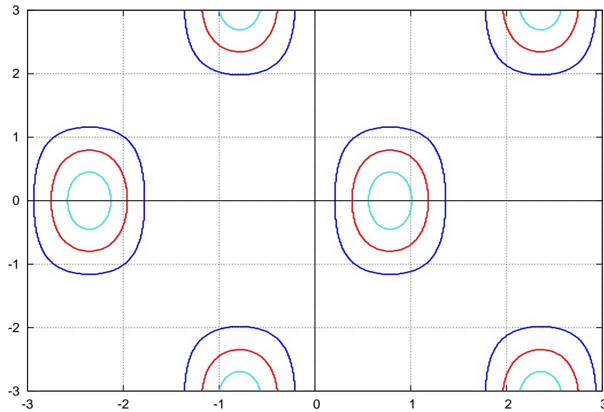

Figure 13: contour plot of $sin(2\,x)\,cos(y)$ using imp()

Of course if we define g, say, to be the expression $sin(2\,x)\,cos(y)$ first, we can use that binding to simplify our call to **imp(...)** :

```
(%i18) g : sin(2*x)*cos(y)$
(%i19) qdraw( imp( [g = 0.4,g = 0.7,g = 0.9] ,x,-3,3,y,-3,3 ) ,
                cut(key) );
```

to achieve the same plot.

We can also use symbols like `%pi`, which will evaluate to a real number, in our horizontal and vertical limit slots, as in:

```
(%i20) qdraw( imp( [g = 0.4,g = 0.7,g = 0.9] ,x,-%pi,%pi,y,-%pi,%pi ) ,
                cut(key) );
```

We need to arrange that the horizontal canvas width is about 1.4 time the vertical canvas height in order that geometrical shapes look closer to reality. For the present plot we simply change the numerical values of the **imp(...)** function (x1,x2) parameters:

```
(%i21) qdraw( imp( [g = 0.4,g = 0.7,g = 0.9] ,x,-4.2,4.2,y,-3,3 ) ,
                cut(key) );
```

which produces a slightly different looking plot:

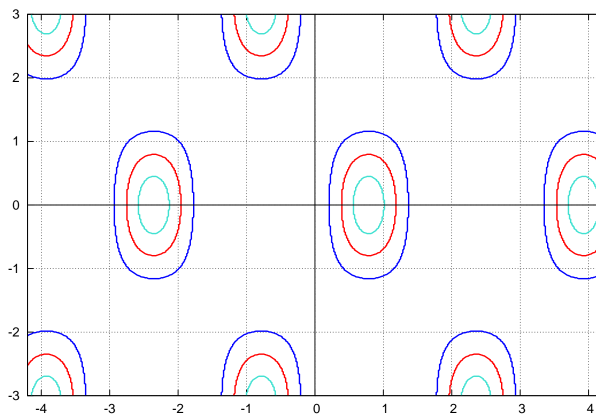

Figure 14: using $(x1, x2) = (-4.2, 4.2)$

5.3 Contour Plots with contour(...)

Since we are talking about contour plots, this is a natural place to give some examples of the **qdraw** package's **contour(...)** function which has two forms:

```
contour( expr,x,x1,x2,y,y1,y2,cvals( v1,v2,...),options )
contour( expr,x,x1,x2,y,y1,y2, crange(n,min,max), options ) .
```

where `expr` is assumed to be a function of (x,y) and the first form allows the setting of `expr` to the supplied numerical values, while the second form allows one to supply the number of contours (n), the minimum value for a contour (min) and the maximum value for a contour (max). If we use the most basic `cvals(...)` form (ignoring options):

```
(%i22) qdraw( contour(g, x,-4.2,4.2, y,-3,3, cvals(0.4,0.7,0.9) ) );
```

we get a "plain jane" contour plot having line width 1, the key, grid, and xy-axes removed, in "black": Since

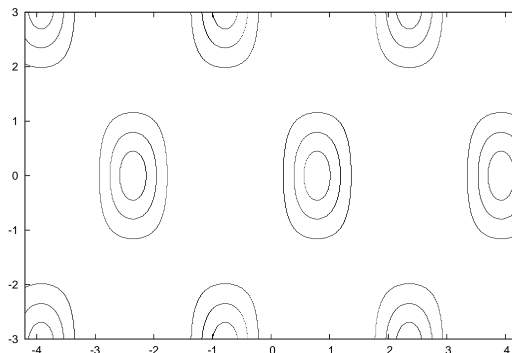

Figure 15: simplest default contour example

the quick plot functions **ex** and **imp** both use the rotating default colors which cannot be turned off, we would have to use the **imp1** function (which we have not yet discussed) with some of its options, to get the same results as the default use of **contour** produces. The available "options" which can be used in any order but after the required first eight arguments, are **lw(n)**, **lc(color)**, and **add(options)**, where the "add options" are any or all of the set `[grid,xaxis,yaxis,xyaxes]`.

Thus the following invocation of **contour**:

```
(%i23) qdraw( contour(g,x,-4.2,4.2,y,-3,3,cvals(0.4,0.7,0.9),
        lw(2),lc(brown) ), ipgrid(15) );
```

produces:

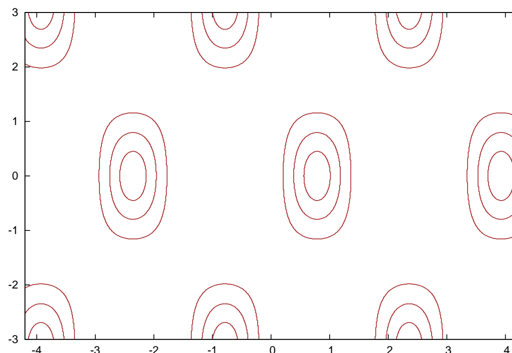

Figure 16: adding lw(2), lc(brown)

We also added the separate **qdraw** function **ipgrid** with argument 15 to over-ride the **qdraw** default value of the **draw2d** parameter `ip_grid_in` . The **draw2d** default for this parameter is 5, which results in some "jaggies" in implicit plots. The default value inside the **qdraw** package is 10, which generally produces smoother plots, but the drawing process takes more time, of course. For our example here, we increased this parameter from 10 to 15 to get a smoother plot at the price of increased drawing time.

Here is an example of using the second, "crange", form of **contour**:

```
(%i24) qdraw( contour(g,x,-4.2,4.2,y,-3,3,crange(4,0.2,0.9),
       lw(2),lc(brown) ), ipgrid(15)  )$
```

which produces the plot:

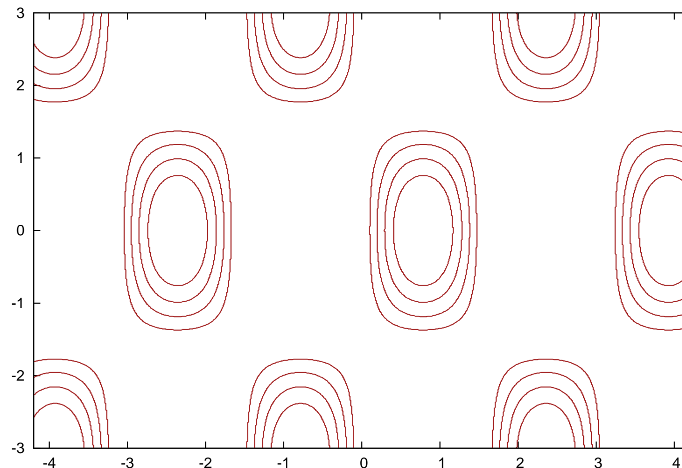

Figure 17: using $crange(4, .2, .9)$

A final example illustrates the **contour** option **add**:

```
(%i25) qdraw( contour(sin(x)*sin(y),x,-2,2,y,-2,2,crange(4,0.2,0.9),
              lw(3),lc(blue),add(xyaxes) ), ipgrid(15) )$
```

with the plot:

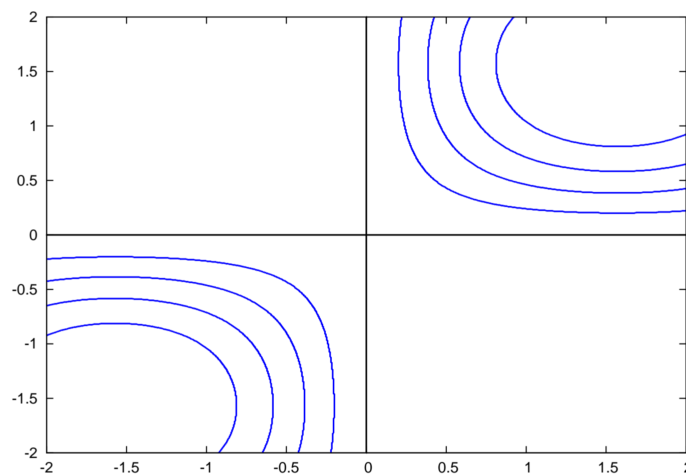

Figure 18: using add(xyaxes)

5.4 Density Plots with qdensity(...)

A type of plot closely related to the contour plot is the density plot, which paints small regions of the graphics window with a variable color chosen to highlight regions where the function of two variables takes on large values. A completely separate density plotting function, **qdensity**, is supplied in the qdraw package. The **qdensity** function in completely independent of the default conventions and syntax associated with the function **qdraw**.

The syntax of **qdensity** is:
`qdensity(expr,[x,x1,x2,dx],[y,y1,y2,dy], options palette(p),pic(..))` , where the two optional arguments are palette(p) and pic(type,filename). The `x` interval `(x1,x2)` is divided into subintervals of size `dx`, and likewise the `y` interval `(y1,y2)` is divided into subintervals of size `dy`.

If the palette(p) option is not present, a default "shades of blue" density plot is drawn (which corresponds to `palette = [1,3,8]`). To use the paletter option, the argument "p" can be either blue, gray, color, or a three element list `[n1,n2,n3]`, where (n1,n2,n3) are positive integers which select functions to apply respectively to red, green, and blue.

To use the pic(...) option, the type is eps, eps_color, jpg, or png, and the filename is a string like "case5a". As usual, use "x" and "y" if `expr` depends explicitly on x and y, or use "u" and "v" if `expr` depends explicitly on u and v, etc.

A simple function of two variables to try is $f(x, y) = x\,y$, which increases from zero at the origin to 1 at (1,1).

```
(%i26) qdensity(x*y,[x,0,1,0.2],[y,0,1,0.2] )$
```

This produces the density plot:

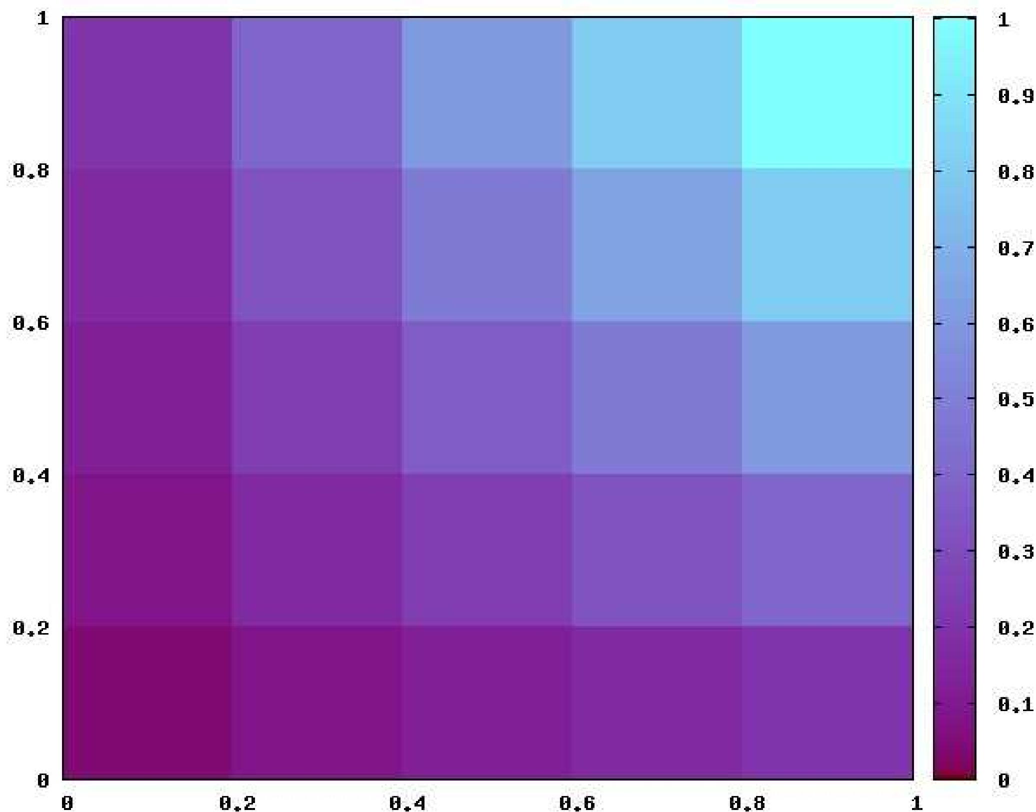

Figure 19: default palette density plot

If we use the gray palette opton

```
(%i27) qdensity(x*y,[x,0,1,0.2],[y,0,1,0.2],
                   palette(gray) )$
```

we get

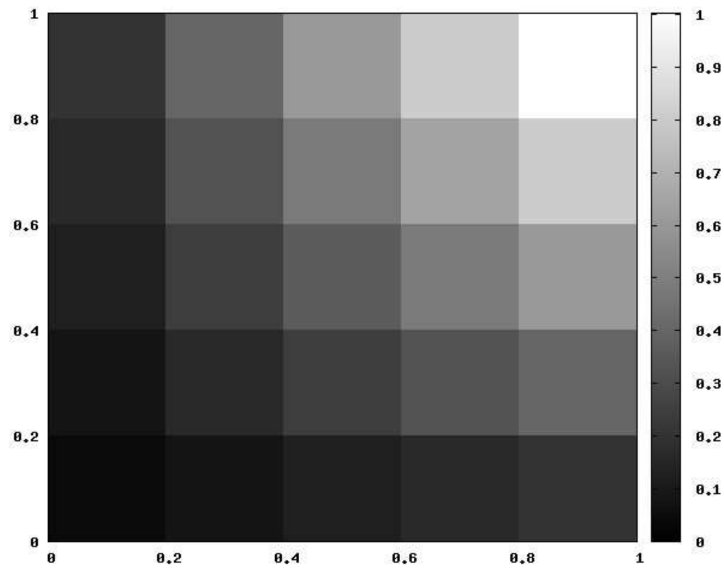

Figure 20: palette(gray) option

while if we use palette(color), we get

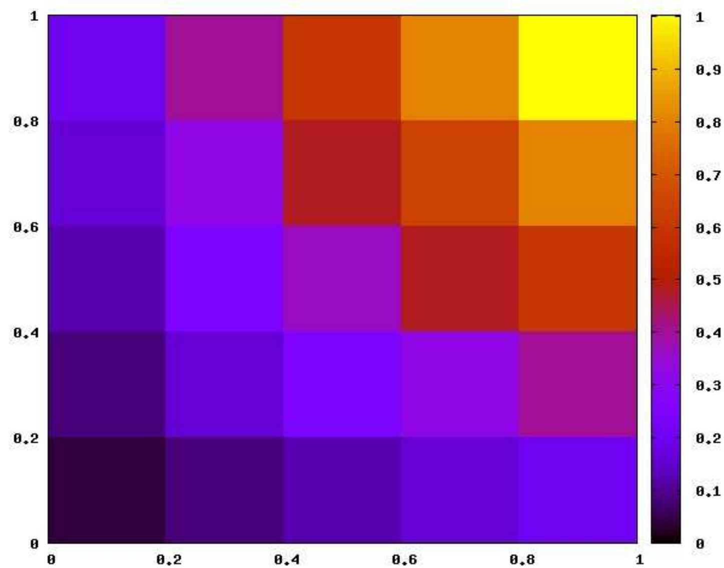

Figure 21: palette(color) option

To get a finer sampling of the function, you should decrease the values of `dx` and `dy` to `0.05` or less. Using the default palette choice with the interval choice `0.05`,

```
(%i28) qdensity(x*y,[x,0,1,0.05],[y,0,1,0.05] )$
```

yields a refined density plot with `20 x 20 = 400` painted rectangular panels.

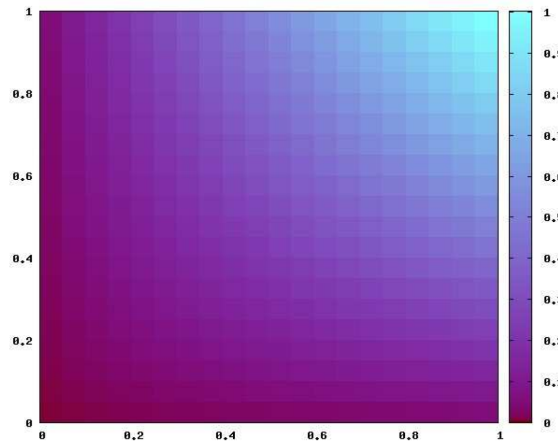

Figure 22: interval set to 0.05

A more interesting function to look at is $f(x, y) = sin(x)\,sin(y)$.

```
(%i29) qdensity(sin(x)*sin(y),[x,-2,2,0.05],[y,-2,2,0.05] )$
```

which yields

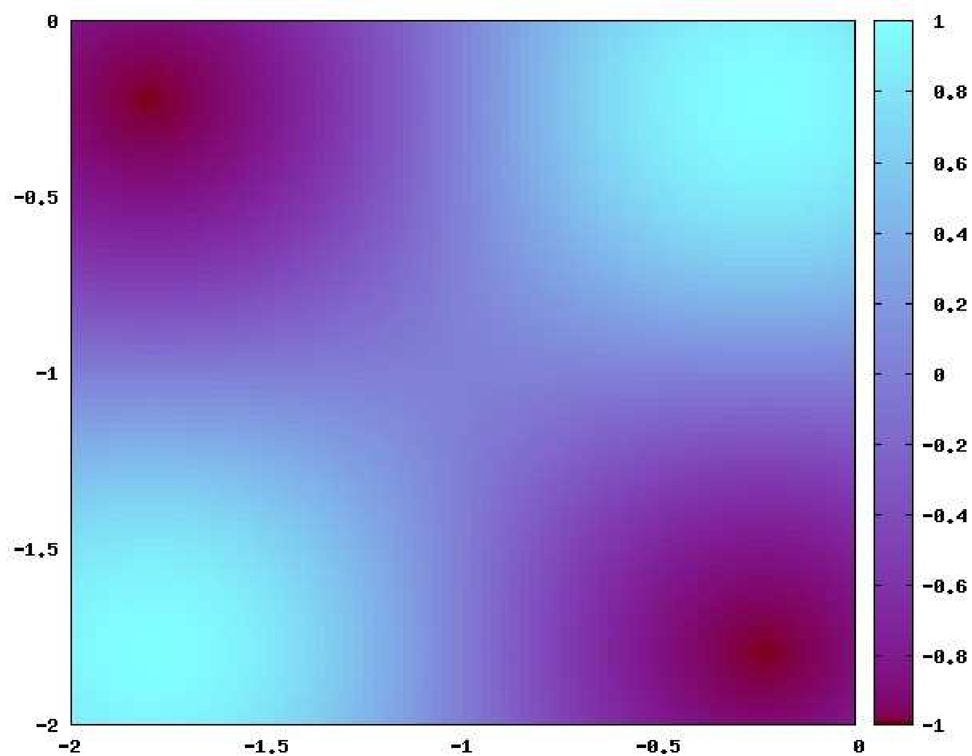

Figure 23: $sin(x)\,sin(y)$

5.5 Explicit Plots with Greater Control: ex1(...)

If we are willing to deal with one explicit function or expression at a time, we get more control over the plot elements if we use the **qdraw** function **ex1(...)**, which has the syntax:

```
ex1( expr, x, x1,x2, lc(c), lw(n),lk(string) ) .
```

As usual, if the expression `expr` is actually a function of `u`, then $x \to u$. The first four arguments are required and must be in the first four slots. The last three arguments are all optional and can be in any order.

Let's illustrate the use of **ex1(...)** by displaying a simple curve and the tangent and normal at one point of the curve. We will use the curve $y = x^2$ with the "slope" $dy/dx = 2\,x$, and construct the tangent line tangent at the point (x_0, y_0):

$$(y - y_0) = m\,(x - x_0)$$

where m is the slope at (x_0, y_0). As we discuss in the next chapter, the normal line through the same point is

$$(y - y_0) = (-1/m)\,(x - x_0).$$

For the point $x_0 = 1, y_0 = 1, m = 2$, the tangent line is $y = 2\,x - 1$ and the normal line is $y = -x/2 + 3/2$.

```
(%i30) qdraw( xr(-1.4,2.8),yr(-1,2),
        ex1(x^2,x,-2,2,lw(5),lc(brown),lk("X^2")),
        ex1(2*x-1,x,-2,2,lw(2),lc(blue),lk("TANGENT")),
        ex1(-x/2 + 3/2,x,-2,2,lw(2),lc(magenta),lk("NORMAL") ) ,
         pts( [ [1,1] ],ps(2),pc(red) ) )$
```

Note that we were careful to force the x-range to be about 1.4 times as great as the y-range (to get the correct geometry of the tangent and normal lines). The resulting plot is:

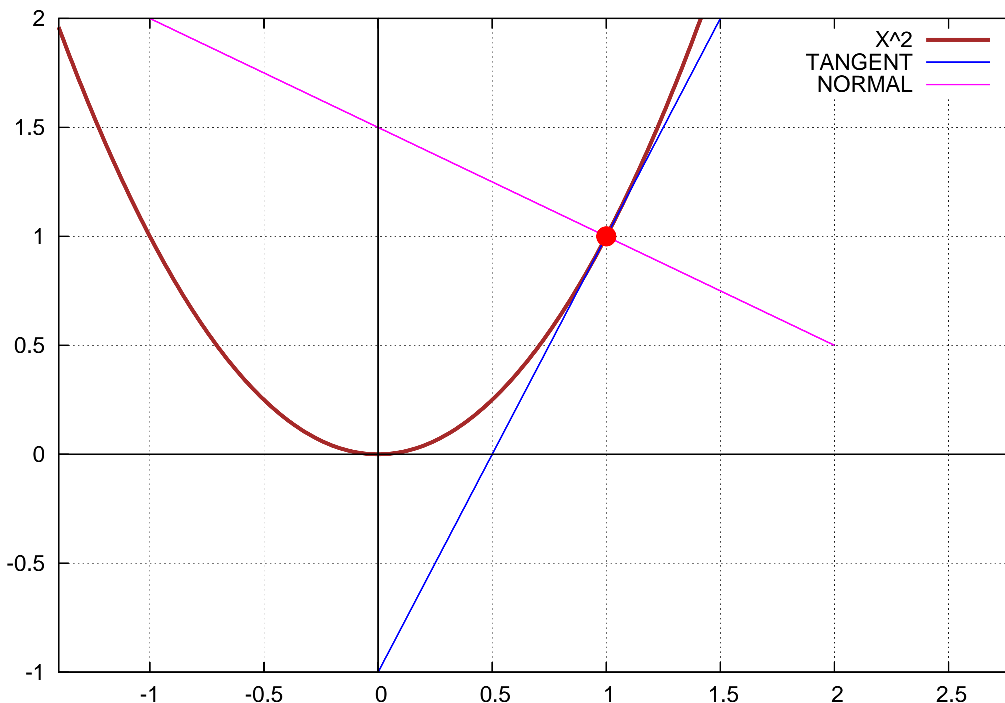

Figure 24: plot using $ex1(...)$

Here we use **ex1** to plot the first few Bessel functions of the first kind $J_n(x)$ for integral n and real x,

```
(%i31) qdraw( ex1(bessel_j(0,x),x,0,20,lc(red),lw(6),lk("bessel_j ( 0, x)") ),
          ex1(bessel_j(1,x),x,0,20,lc(blue),lw(5),lk("bessel_j ( 1, x)")),
          ex1(bessel_j(2,x),x,0,20,lc(brown),lw(4),lk("bessel_j ( 2, x)") ),
          ex1(bessel_j(3,x),x,0,20,lc(green),lw(3),lk("bessel_j ( 3, x)") ) )$
```

which produces the plot:

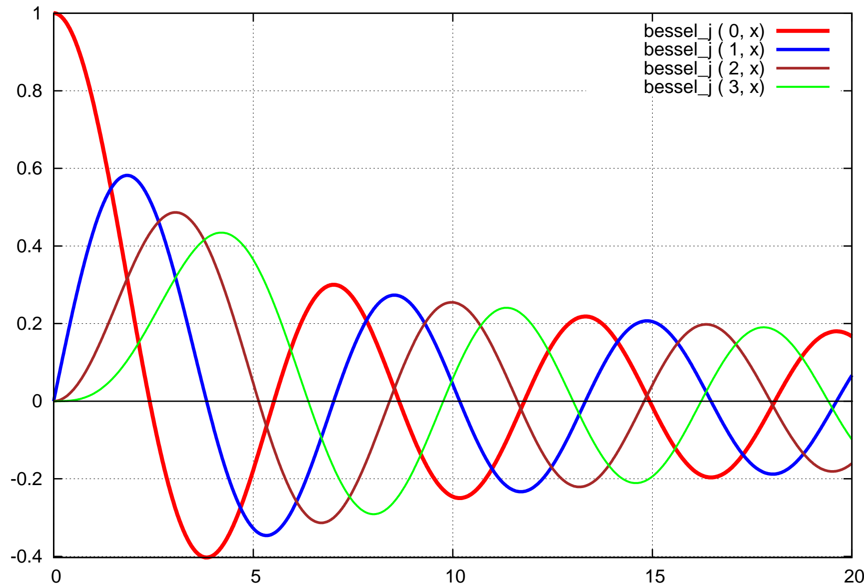

Figure 25: $J_n(x)$

Here is a plot of $J_0(\sqrt{x})$:

```
(%i32) qdraw(line(0,0,50,0,lc(red),lw(2) ),
             ex1(bessel_j(0, sqrt(x)),x,0,50 ,lc(blue),
              lw(7),lk("J0( sqrt(x) )") ) )$
```

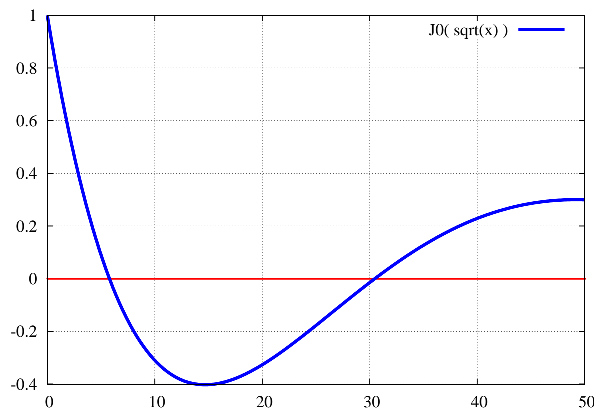

Figure 26: $J_0(\sqrt{x})$

We chose to emphasize the axis $y = 0$ with a red line supplied by another of the **qdraw** functions, **line**, which we will discuss later in the section on geometric figures. Placing the **line** element before **ex1(..)** causes the curve to write ”over” the line, rather than the reverse.

5.6 Explicit Plots with ex1(...) and Log Scaled Axes

The name "log plot" usually refers to a plot of $\ln(y)$ vs x using linear graph paper, which is equivalent to a plot of y vs x on graph paper which uses a "logarithmic scale" on the vertical axis. Given an expression g depending on x, you can either use the syntax `qdraw( ex1( log(g),x,x1,x2 ), other options )` to generate such a "log plot" or `qdraw( ex1(g, x, x1, x2), log(y) , other options ).`

Let's show the differences using the function $f(x) = x\,e^{-x}$, but using an expression called g rather than a Maxima function.

```
(%i33) g : x*exp(-x)$
(%i34) qdraw( ex1( log(g),x,0.001,10, lc(red) ),yr(-8,0)  )$
```

which displays the plot

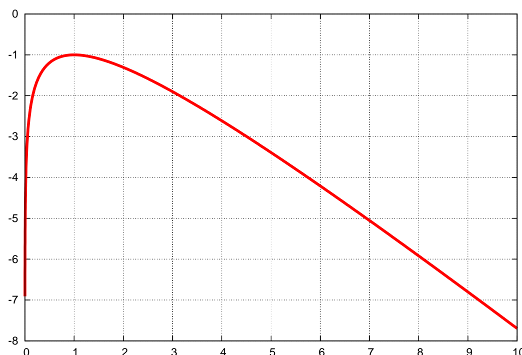

Figure 27: Linear Graph Paper Plot of $\ln(g)$

The numbers on the vertical axis correspond to values of $\ln(g)$. Since g is singular at $x = 0$, we have avoided that region by using $x_1 = 0.001$.

The second way to get a "log plot" of g is to request "semi-log" graph paper which has the vertical axis marked using a logarithmic scale for the values of g. Using the **log(y)** option of the **qdraw** function, we use:

```
(%i35) qdraw( ex1(g, x, 0.001,10,lc(red) ),
                yr(0.0001, 1), log(y) )$
```

The **yr(y1,y2)** option takes into account the numerical limits of g over the x interval requested. The minimum value of g is 0.005 which occurs at $x = 10$. The maximum value of g is about 0.37. The resulting plot is:

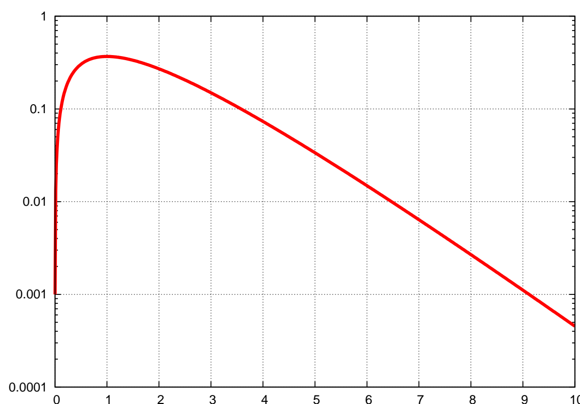

Figure 28: Log Paper Plot of g

The name "log-linear plot" can be used to mean "x axis marked with a log scale, y axis marked with a linear scale". Using the same function, we generate this plot by using the **log(x)** option to **qdraw**:

```
(%i36) qdraw( ex1(g, x, 0.001,10,lc(red),lw(7) ),
                 yr(0,0.4), log(x) )$
```

This generates the plot

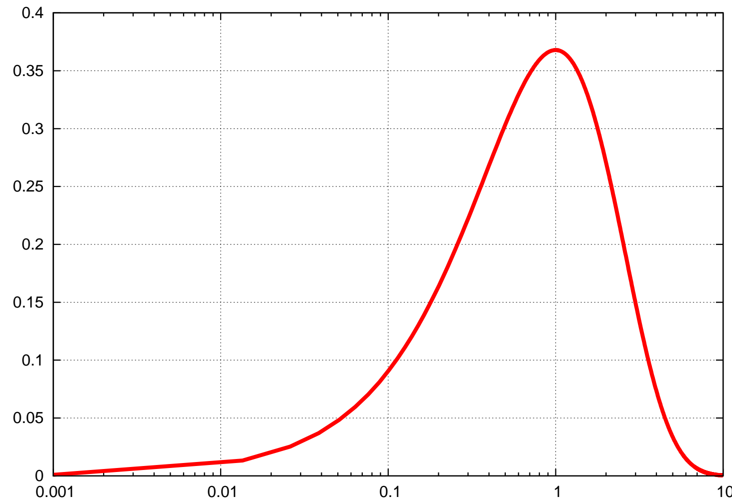

Figure 29: Log-Linear Plot of g

Scientists and engineers normally like to use a log scaled axis for a variable which varies over many powers of ten, which is not the case for our example.

Finally, we can request "log-log paper" which has both axes marked with a log scale, by using the **log(xy)** option to **qdraw**.

```
(%i37) qdraw( ex1(g, x, 0.001,10,lc(red) ),
                  yr(0.0001,1), log(xy) )$
```

which produces

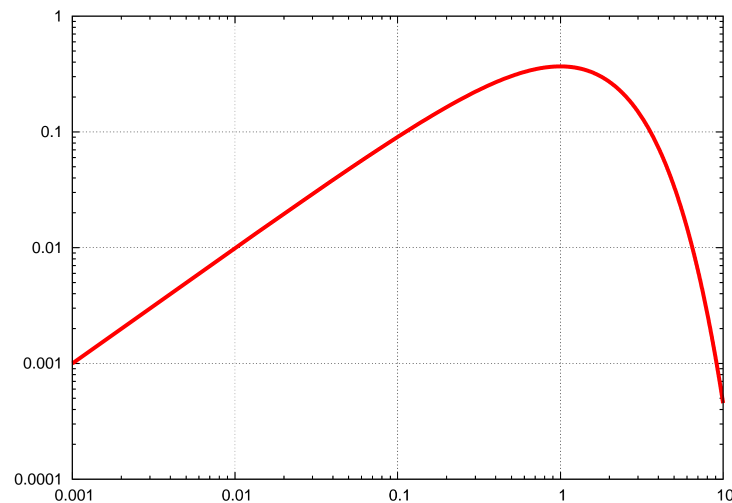

Figure 30: Log-Log Plot of g

5.7 Data Plots with Error Bars: pts(...) and errorbars(...)

In Chapter One of Maxima by Example, Section 1.5.8, we created a data file called "fit1.dat", which can be downloaded from the author's webpage. We will use that data file, together with "fit2.dat", also available, to illustrate making simple data plots using the **qdraw** functions **pts(...)** and **errorbars(...)**. The syntax of **pts(...)** is:

`pts( pointlist, pc(c), ps(s), pt(t), pj(lw), pk(string) )`

The only required argument is the first argument "pointlist" which has the form:

`[ [x1,y1], [x2,y2], [x3,y3],...] .`

The remaining arguments are all optional and may be entered in any order following the first required argument.

The optional argument `pc(c)` overrides the default color (black), for example, `pc(red)`.

The optional argument `ps(s)` overrides the default size (3), and an example is `ps(2)`.

The optional argument `pt(t)` overrides the default type (7, which is the integer used for filled_circle: see the Maxima manual index entry for "point_type"); an example would be `pt(8)`, which would use an open "up_triangle" instead of a filled circle.

The optional argument `pj(lw)`, if present, will cause the points provided by the nested list "pointlist" to be joined using a line whose width is given by the argument of `pj`; an example is `pj(2)` which would set the line width to the value 2.

The optional argument `pk(string)` provides text for a key entry for the set of points represented by pointlist; as example is `pk("case x^2")`.

Before making the data plot, let's look at the data file contents from inside Maxima:

```
(%i38) printfile("fit1.dat")$
1 1.8904
2 3.0708
3 3.9215
4 5.1813
5 5.9443
6 7.0156
7 7.8441
8 8.8806
9 9.8132
10 11.129
```

We next use Maxima's **read_nested_list** function to create a list of data points from the data file.

```
(%i39) plist : read_nested_list("fit1.dat");
(%o39) [[1, 1.8904], [2, 3.0708], [3, 3.9215], [4, 5.1813], [5, 5.9443],
   [6, 7.0156], [7, 7.8441], [8, 8.880599999999999], [9, 9.8132], [10, 11.129]]
```

The most basic plot of this data uses the **pts(...)** function defaults:

```
(%i40) qdraw( pts(plist) )$
```

which produces:

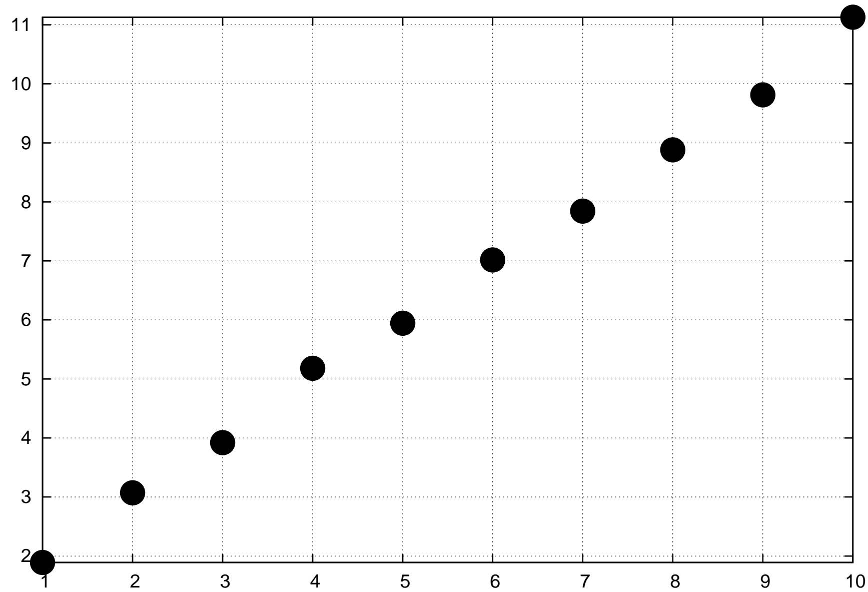

Figure 31: Using pts(...) Defaults

We can use the **qdraw** functions **xr(...)** and **yr(...)** to override the default range selected by **draw2d**, and decrease the point size:

```
(%i41) qdraw( pts(plist, ps(2)), xr(0,12),yr(0,15) )$
```

with the result:

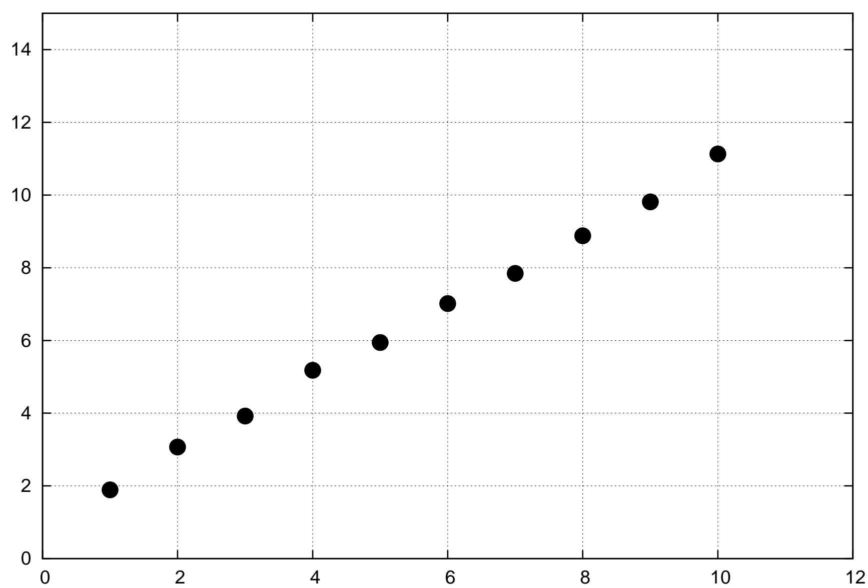

Figure 32: Adding ps(2), xr(..), yr(..)

Now we add color and a key string, as well as simple error bars corresponding to an assumed uncertainty of the `y` value of plus or minus `1` for all the data points.

```
(%i42) qdraw( pts(plist,pc(blue),pk("fit1"), ps(2)), xr(0,12),yr(0,15),
          key(bottom), errorbars( plist, 1) )$
```

which looks like:

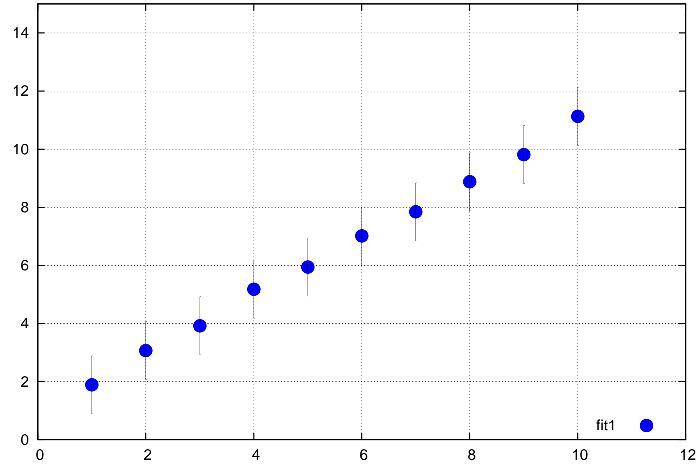

Figure 33: Adding pc(blue) and Simple Error Bars

The default error bar line width of `1` is almost too small to see, so we thicken the error bars and add color

```
(%i43) qdraw( pts(plist,pc(blue),pk("fit1"), ps(2)), xr(0,12),yr(0,15),
          key(bottom), errorbars( plist, 1, lw(3),lc(red) )   )$
```

with the result:

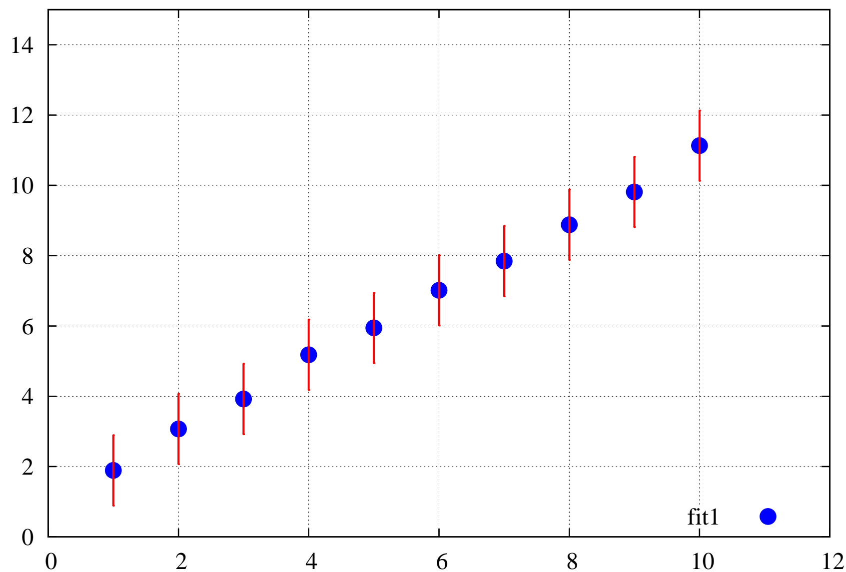

Figure 34: Adding lw(3), lc(red) to errorbars(...)

The difference in the fonts is due to my using `pic(eps, "ch5p27h", font("Times-Roman",18) )` to create the eps graphic instead of just `pic(eps, "ch5p27h" )` as another argument to **qdraw**.

If the data set has individual uncertainties in the y value, we create a list dyl, say, of the values dy1, dy2, dy3, ... and use the syntax:

```
errorbars( pointlist, dylist, lw(n), lc(c) )
```

Here is an example:

```
(%i44) dyl : [0.2,0.3,0.5,1.5,0.8,1,1.4,1.8,2,2];
(%o44)           [0.2, 0.3, 0.5, 1.5, 0.8, 1, 1.4, 1.8, 2, 2]
(%i45) map(length,[plist,dyl] );
(%o45)                                [10, 10]
(%i46) qdraw( pts(plist,pc(blue),pk("fit1"), ps(2)), xr(0,12),yr(0,15),
          key(bottom), errorbars( plist, dyl, lw(3),lc(red) )  )$
```

with the result

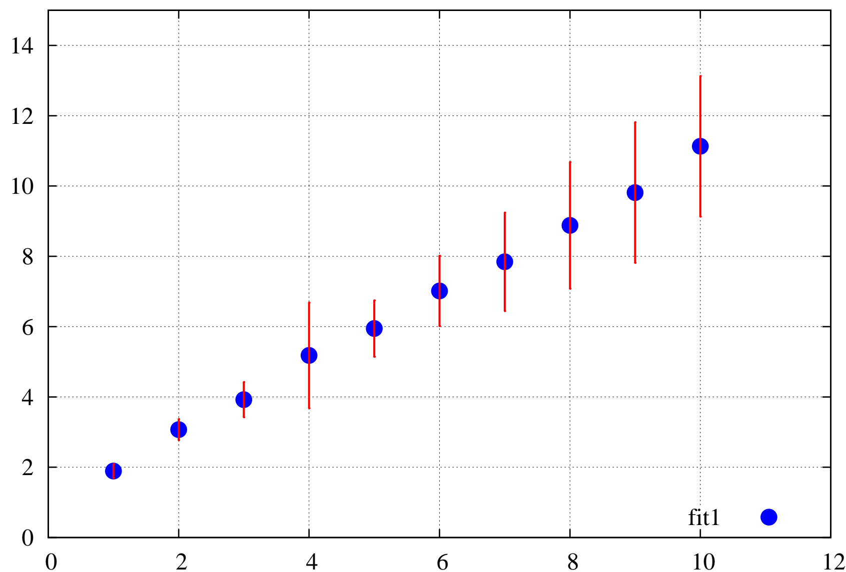

Figure 35: Using a list of dy values with errorbars(..)

We now repeat the least squares fit of this data which we carried out in Chapter 1. See our discussion there for an explanation of what we are doing here.

```
(%i47) display2d:false$
(%i48) pmatrix : apply( 'matrix, plist );
(%o48) matrix([1,1.8904],[2,3.0708],[3,3.9215],[4,5.1813],[5,5.9443],
              [6,7.0156],[7,7.8441],[8,8.880599999999999],[9,9.8132],
              [10,11.129])
(%i49) load(lsquares);
(%o49) "C:/PROGRA~1/MAXIMA~4.0/share/maxima/5.15.0/share/contrib/lsquares.mac"
(%i50) soln : (lsquares_estimates(pmatrix,[x,y],y=a*x+b,
                   [a,b]), float(%%) );
(%o50) [[a = 0.99514787679748,b = 0.99576667381004]]
(%i51) [a,b] : (fpprintprec:5, map( 'rhs, soln[1] ) )$
(%i52) [a,b];
(%o52) [0.995,0.996]
(%i53) qdraw( pts(plist,pc(blue),pk("fit1"), ps(2)), xr(0,12),yr(0,15),
          key(bottom), errorbars( plist, dyl, lw(3),lc(red) ),
         ex1( a*x + b,x,0,12, lc(brown),lk("linear fit") )  )$
```

We use the **qdraw** function **ex1(...)** to add the line $f(x) = a\,x + b$ to the data plot. The resulting plot with the least squares fit added is then:

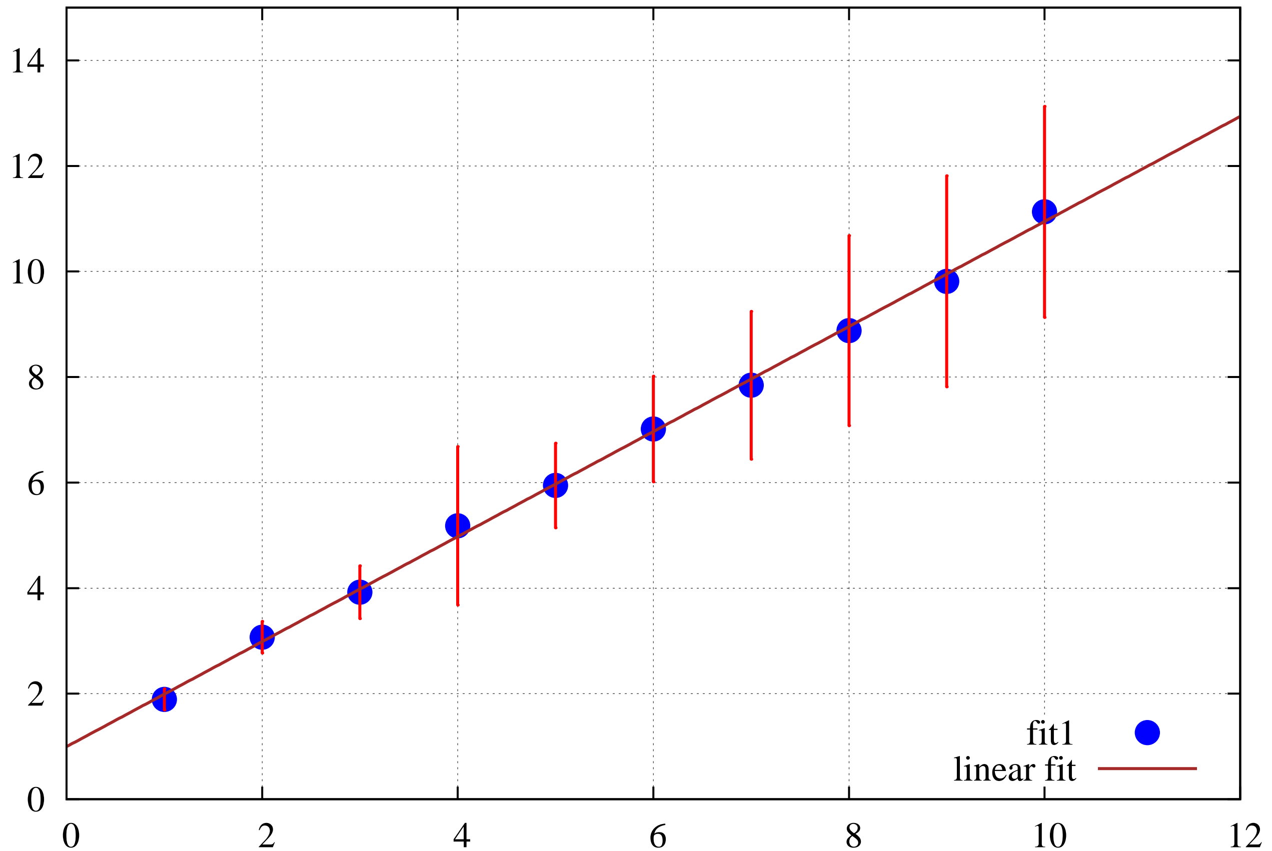

Figure 36: Adding the Linear Fit Line

Now we add the data in the file "fit2.dat":

```
(%i54) printfile("fit2.dat");
1 0.9452
2 1.5354
3 1.9608
4 2.5907
5 2.9722
6 3.5078
7 3.9221
8 4.4403
9 4.9066
10 5.5645
(%o54) "fit2.dat"
(%i55) p2list: read_nested_list("fit2.dat");
(%o55) [[1,0.945],[2,1.5354],[3,1.9608],[4,2.5907],[5,2.9722],[6,3.5078],
        [7,3.9221],[8,4.4403],[9,4.9066],[10,5.5645]]
(%i56) qdraw( pts(plist,pc(blue),pk("fit1"), ps(2)), xr(0,12),yr(0,15),
          key(bottom), errorbars( plist, dyl, lw(3),lc(red) ),
         ex1( a*x + b,x,0,12, lc(brown),lk("linear fit 1") ),
         pts(p2list, pc(magenta),pk("fit2"),ps(2)),
         errorbars( p2list,0.5,lw(3) ) )$
```

Here is the plot with the second data set:

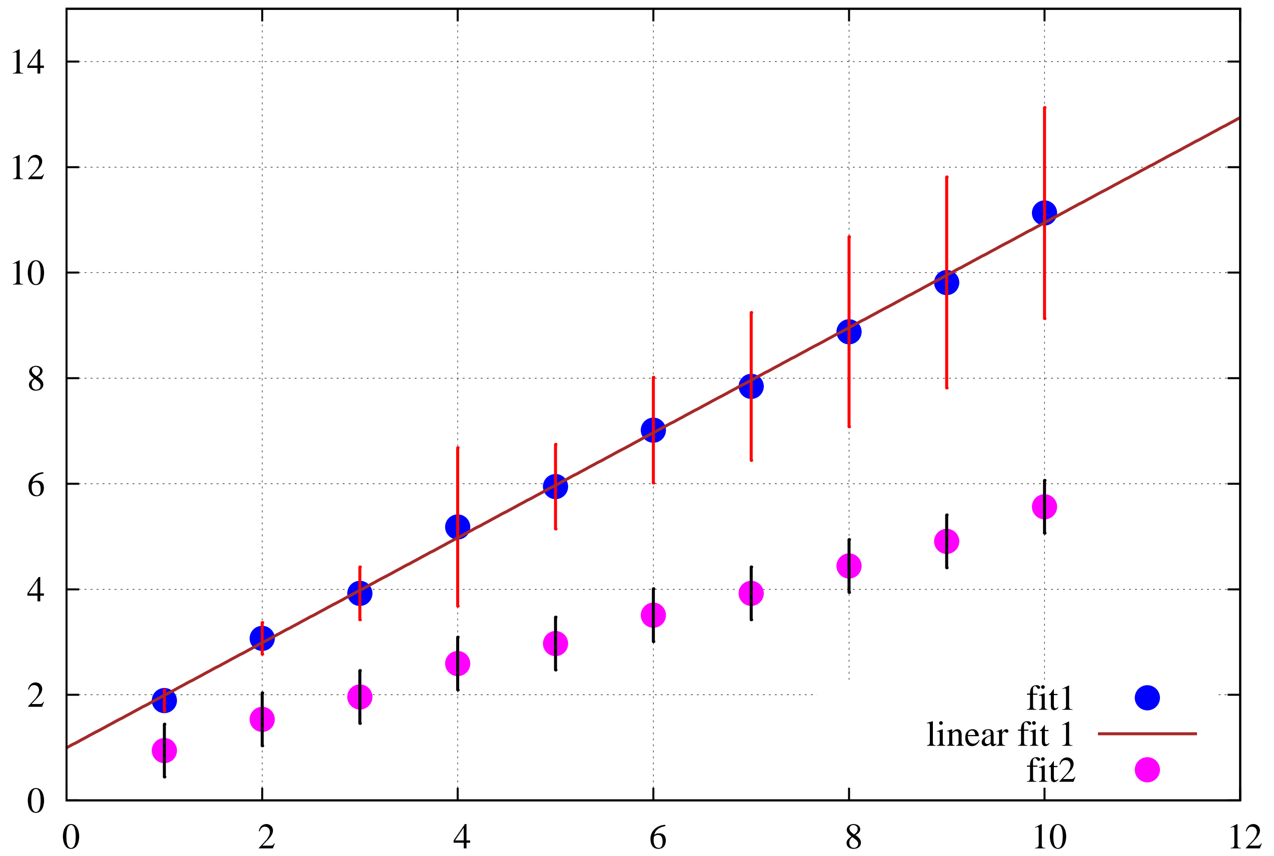

Figure 37: Adding the Second Set of Data

We could then find the least squares fit to the data set 2 and again use the function **ex1(...)** to add that fit to our plot, and add any other features desired.

5.8 Implicit Plots with Greater Control: imp1(...)

If we are willing to deal with one implicit equation of two variables at a time, we get more control over the plot elements if we use the **qdraw** function **imp1(...)**, which has the syntax:

```
imp1( eqn, x, x1,x2, y, y1,y2, lc(c), lw(n),lk(string) ) .
```

As usual, if the equation `eqn` is actually a function of the pair of variables `u` and `v`, then let $x \to u$, and $y \to v$. The first seven arguments are required and must be in the first seven slots. The last three arguments are all optional and can be in any order.

Let's illustrate the use of **imp1(...)** by displaying a translated and rotated ellipse, together with the rotated x and y axes. In the following, `eqn1` describes the rotated ellipse, `eqn2` describes the rotated x axis, and `eqn3` describes the rotated y axis. The angle of rotation is about $63.4\,\deg$ (counter clockwise), which corresponds to $\tan\phi = 2$. Notice that we take care to get the x-axis range about 1.4 times the y-axis range, in order to get the geometry approximately right.

```
(%i1) eqn1 : 5*x^2 + 4*x*y + 8*y^2 - 16*x + 8*y - 16 = 0$
(%i2) eqn2 : y+1 = 2*(x-2)$
(%i3) eqn3 : y+1 = -(x-2)/2$
(%i4) qdraw( imp1(eqn1,x,-2,6.4,y,-4,2,lc(red),lw(6),lk("ELLIPSE")),
              imp1(eqn2,x,-2,6.4,y,-4,2,lc(blue),lw(4),lk("ROT X AXIS")),
              imp1(eqn3,x,-2,6.4,y,-4,2,lc(brown),lw(4),lk("ROT Y AXIS") ),
          pts([ [2,-1] ],ps(2),pc(magenta),pk("TRANSLATED ORIGIN") ) )$
```

We get the following figure, if we increase the font size,

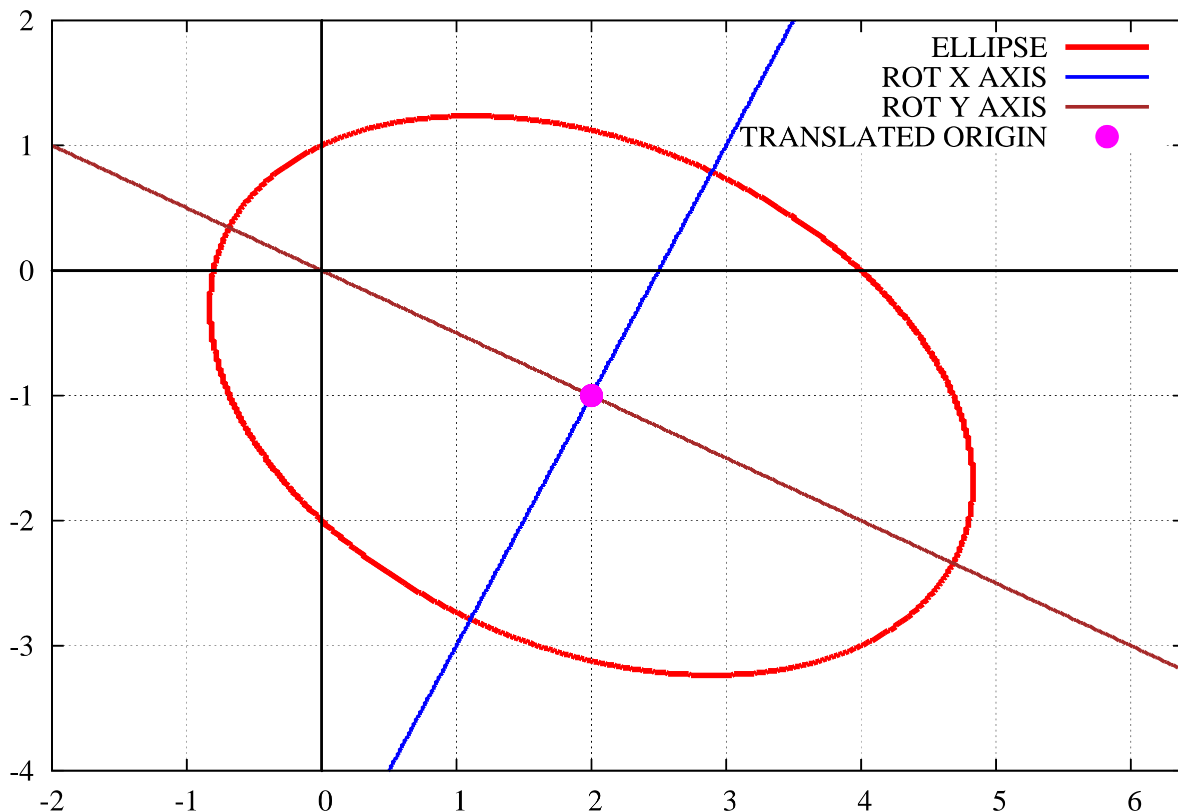

Figure 38: Rotated and Translated Ellipse

As a second example with **imp1** we make a simple plot based on the equation $y^3 = x^2$.

```
(%i5) qdraw( imp1(y^3=x^2,x,-3,3,y,-1,3,lw(10),lc(dark-blue)) )$
```

which produces the plot:

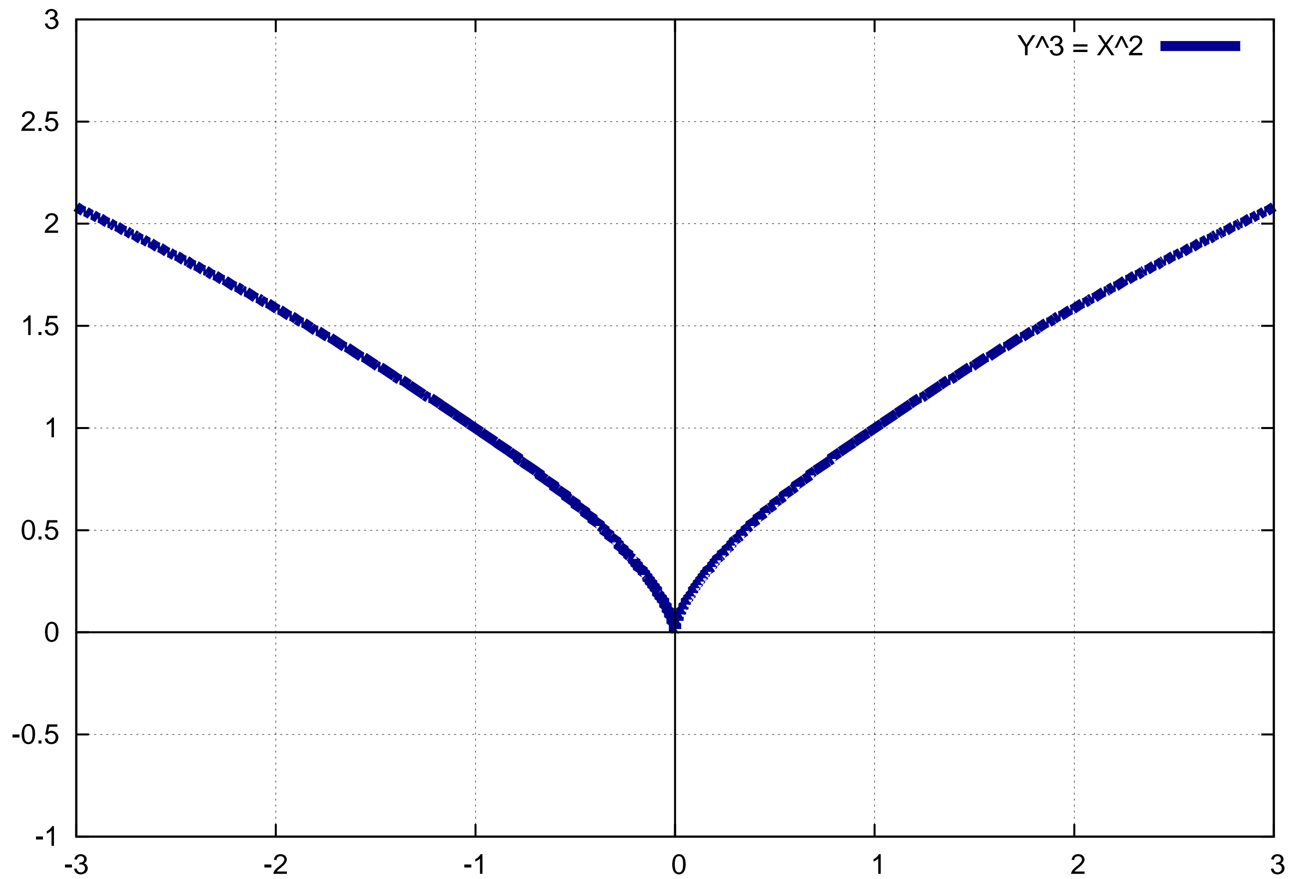

Figure 39: Implicit Plot of $y^3 = x^2$

Notice that you can use hyphenated color choices (see Maxima color index) without the double quotes, or with the double quotes.

5.9 Parametric Plots with para(...)

The **qdraw** function **para** can be used to draw parametric plots and has the syntax
`para(xofu,yofu,u,u1,u2,lw(n),lc(c),lk(string) )`
where, as usual, the line width, line color, and key string are optional and can be in any order. The parameter "u" can, of course, be any symbol.

A simple example, in which we use "t" for the parameter, and let the x coordinate corresponding to some value of t be $sin(t)$, and let the y coordinate corresponding to that same value of t be $sin(2\,t)$ is:

```
(%i6) qdraw(xr(-1.5,2),yr(-2,2),
         para(sin(t),sin(2*t),t,0,2*%pi ) ,
         pts( [ [sin(%pi/8),sin(%pi/4)] ],ps(2),pc(blue),pk("t = pi/8")),
         pts( [ [1,0] ],ps(2),pc(red),pk("t = pi/2")) )$
```

produces the plot:

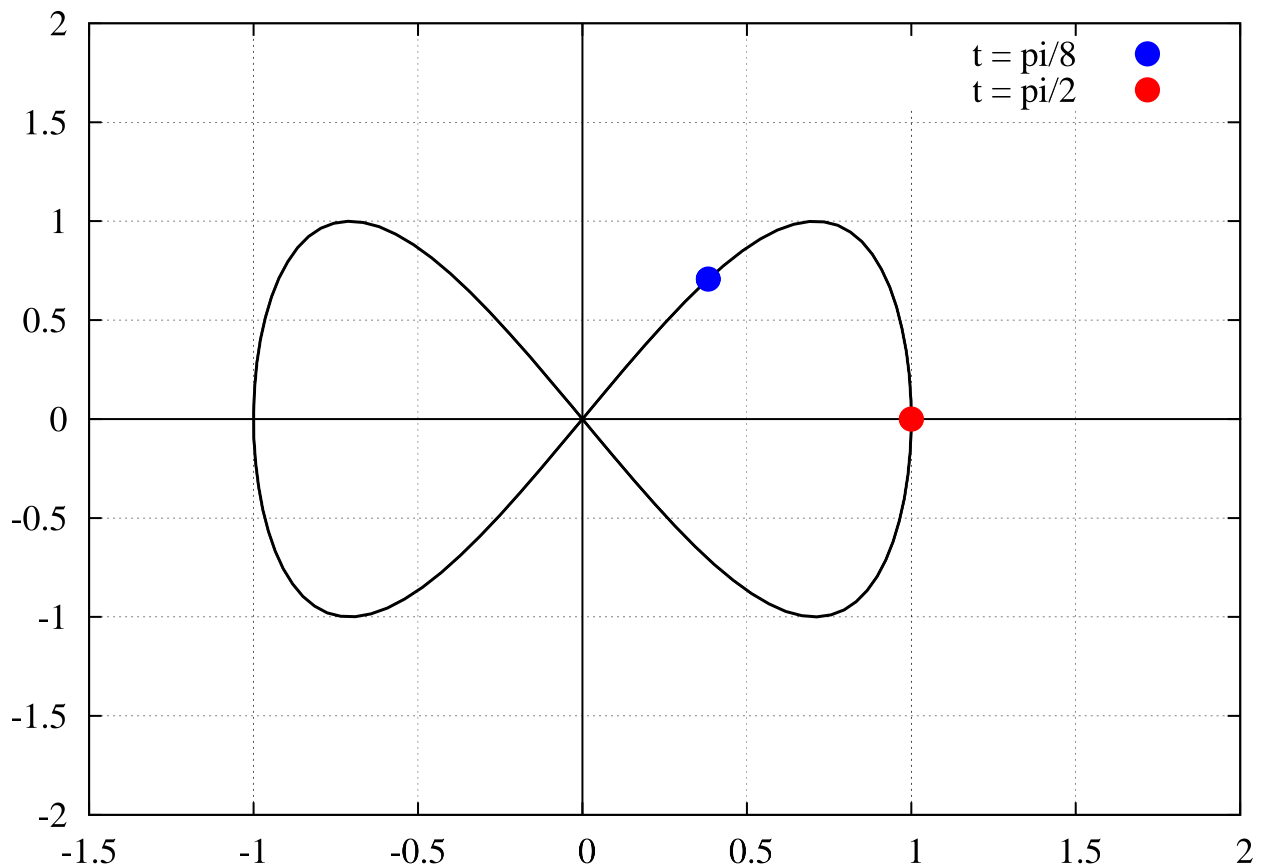

Figure 40: $x = \sin(t), y = \sin(2\,t)$

A second example of a parametric plot has u as the parameter, $x = 2\,cos(u)$, and $y = u^2$:

```
(%i7) qdraw(xr(-3,4),yr(-1,40), para(2*cos(u),u^2,u,0,2*%pi) ,
        pts([ [2,0] ],ps(2),pc(blue),pk("u = 0")),
        pts( [ [0,(%pi/2)^2] ],ps(2), pc(red), pk("u = pi/2")),
        pts([ [-2,%pi^2]],ps(2),pc(green),pk("u = pi")),
        pts( [[0,(3*%pi/2)^2]],ps(2),pc(magenta),pk("u = 3*pi/2")) )$
```

which yields the plot:

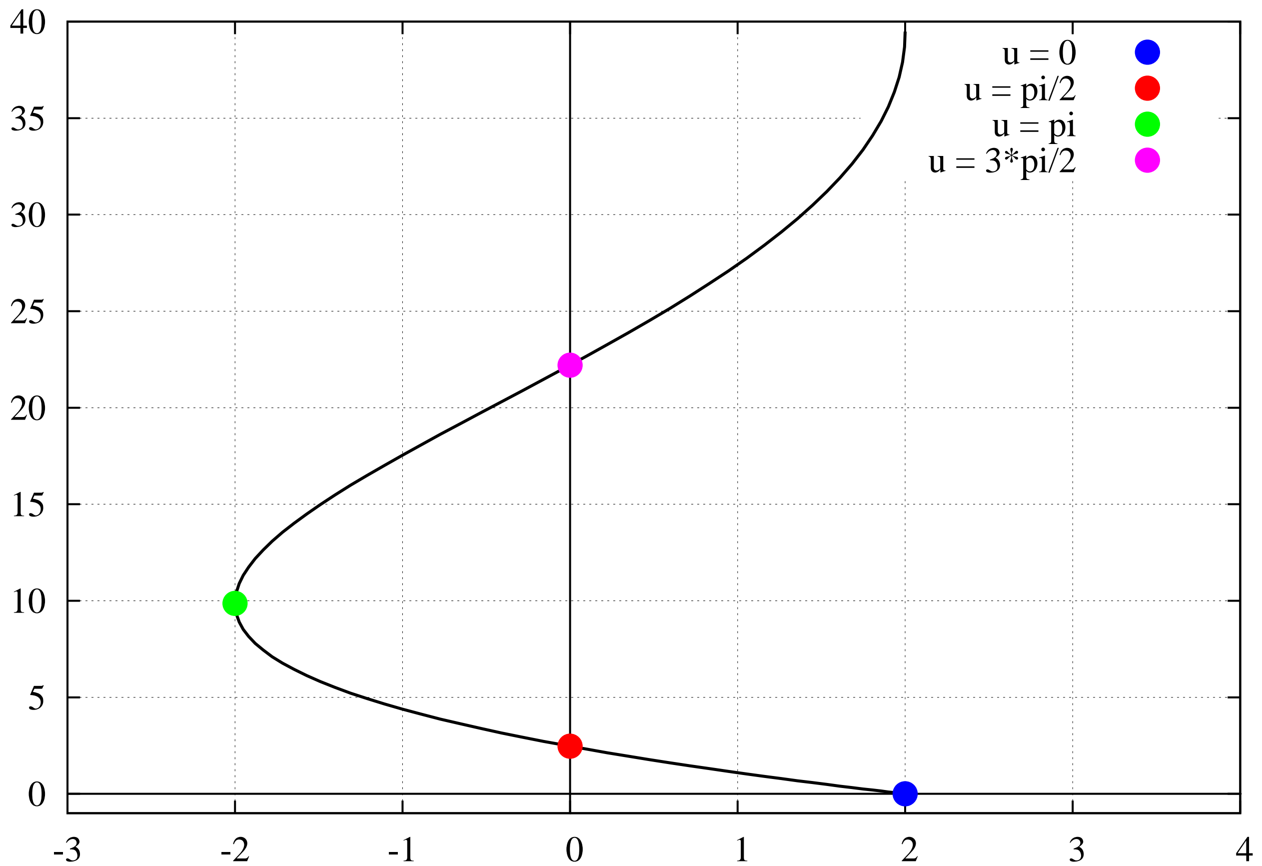

Figure 41: $x = 2\,cos(u), y = u^2$

5.10 Polar Plots with polar(...)

A "polar plot" plots the points $(x = r(\theta)\ \cos(\theta), y = r(\theta)\ \sin(\theta))$, where the function $r(\theta)$ is supplied.

The **qdraw** function **polar** has the syntax:

```
polar( roftheta, theta, th1,th2, lc(c), lw(n), lk(string) )
```

where theta, th1, and th2 are in radians, and the last three arguments are optional.

A simple example is provided by the hyperbolic spiral $r(\theta) = 10/\theta$.

```
(%i8) qdraw( polar(10/t,t,1,3*%pi,lc(brown),lw(5)),nticks(200),
            xr(-4,6),yr(-3,9),key(bottom) ,
         pts( [[10*cos(1),10*sin(1)]],ps(3),pc(red),pk("t = 1 rad")),
       pts([[5*cos(2),5*sin(2)]],ps(3),pc(blue),pk("t = 2 rad") ),
                line(0,0,5*cos(2),5*sin(2)) )$
```

which looks like:

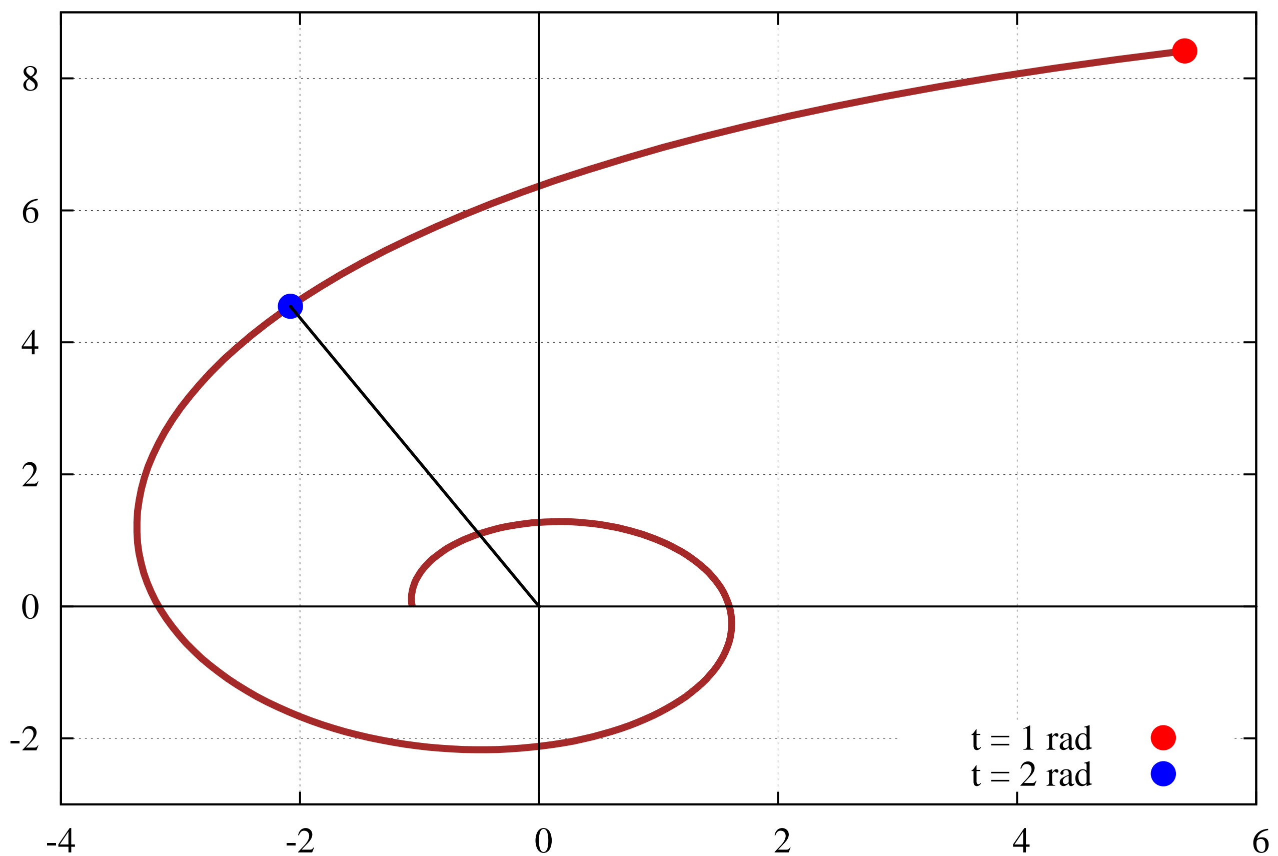

Figure 42: Polar Plot with $r = 10/\theta$

5.11 Geometric Figures: line(...)

The **qdraw** function **line** has the syntax:

```
line( x1,y1,x2,y2, lc(c), lw(n), lk(string) )
```

which draws a line from `(x1,y1)` to `(x2,y2)`. The last three arguments are optional and can be in any order after the first four arguments.

For example, `line(0,0,1,1,lc(blue),lw(6),lk("radius") )` will draw a line from $(0,0)$ to $(1,1)$ in blue with line width 6 and with a key entry with the text 'radius'. The defaults are color black, line width 3, and no key entry.

```
(%i9) qdraw( line(0,0,1,1) )$
```

produces the default line with **draw2d**'s default range:

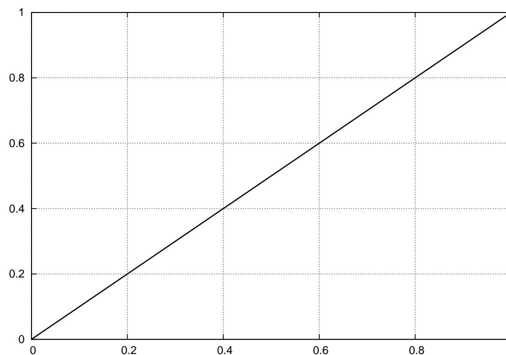

Figure 43: Default line(..)

Adding some options and extending the canvas range in both directions

```
(%i10) qdraw( line(0,0,1,1,lc(blue),lw(6),lk("radius") ),
              xr(0,2),yr(0,2),key(bottom),
              pts([ [1,1] ] ,pc(red),pk("point")) )$
```

produces a line to a point marked in red:

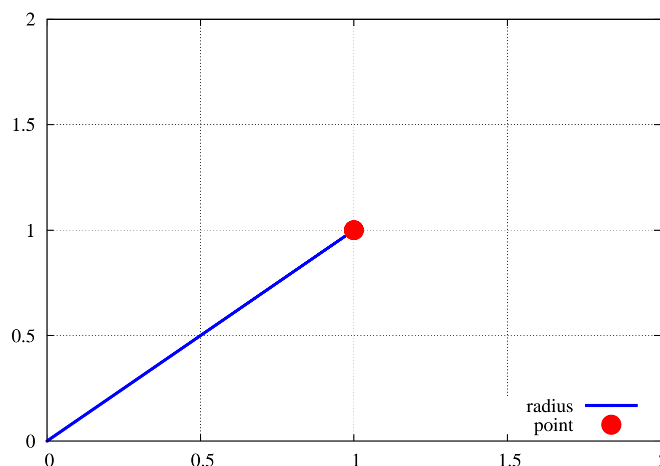

Figure 44: Adding options to line(..)

Here we define a Maxima function ”doplot1(n)” in a file ”doplot1.mac” which has the following contents:

```
/* file doplot1.mac  */

disp("doplot1(nlw)")$

doplot1(nlw) := block([cc,qlist,x,val,i ],
     /* list of 20 single name colors  */
     cc : [aquamarine,beige,blue,brown,cyan,gold,goldenrod,green,khaki,
            magenta,orange,pink,plum,purple,red,salmon,skyblue,turquoise,
            violet,yellow ],
     qlist : [ xr(-3.3,3) ],
     for i thru length(cc) do (
       x : -3.3 + 0.3*i,
       val : line( x,-1,x,1, lc( cc[i] ),lw(nlw) ),
       qlist : append(qlist, [val] )
     ),
     qlist : append( qlist,[ cut(all) ] ),
     apply('qdraw, qlist)
    )$
```

Here is a record of loading and using the function defined to produce a series of vertical colored lines.

```
(%i11) load(doplot1);
                                  doplot1(nlw)
(%o11)                         c:/work2/doplot1.mac
(%i12) doplot1(20);
```

which produces the graphic (note use of **cut**(all) to get a blank canvas):

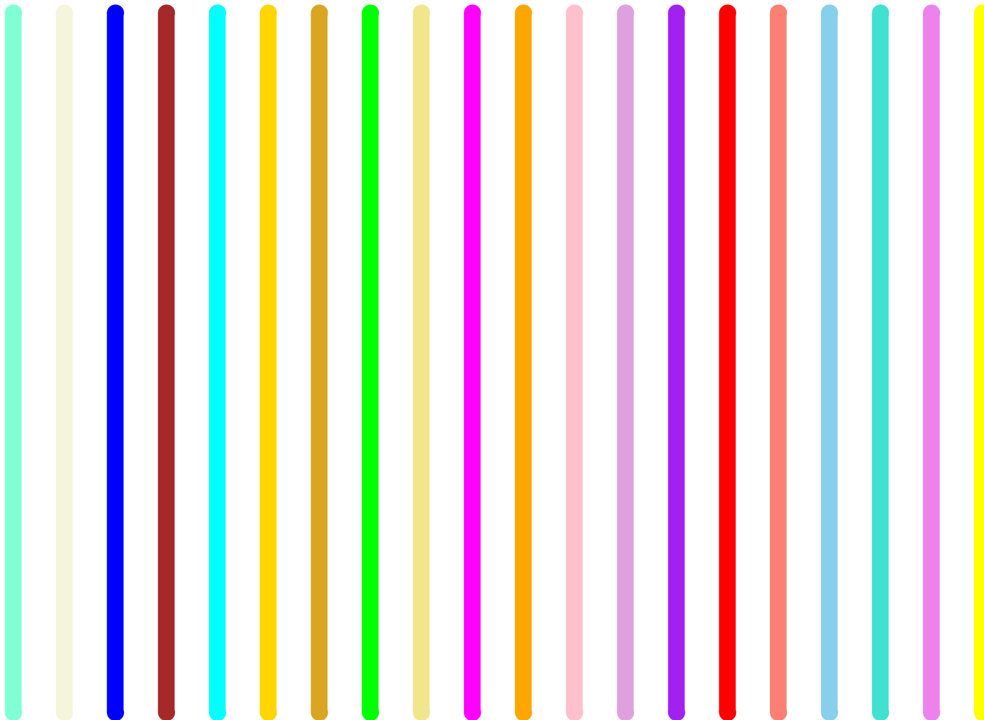

Figure 45: Using line(...) to Display Colors

5.12 Geometric Figures: rect(...)

The **qdraw** function **rect** has the syntax:

```
    rect( x1,y1,x2,y2, lc(c), lw(n), fill(c)  )
```

which will draw a rectangle with opposite corners `(x1,y1)` and `(x2,y2)`. The last three arguments are optional; without them the rectangle is drawn in black with line thickness 3 and with no fill color. An example is `rect(0,0,1,1,lc(brown),lw(2),fill(khaki) )` . We start with the basic rectangle call:

```
(%i13) qdraw( xr(-1,2),yr(-1,2),rect(0,0,1,1) )$
```

with the result

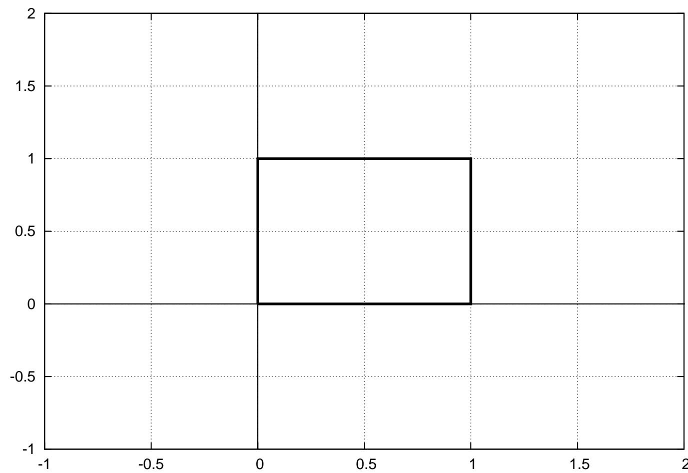

Figure 46: Default rect(0,0,1,1)

We now add some color, thickness and fill:

```
(%i14) qdraw( xr(-1,2),yr(-1,2),
            rect(0,0,1,1,lw(5),lc(brown),fill(khaki) ) )$
```

with the output:

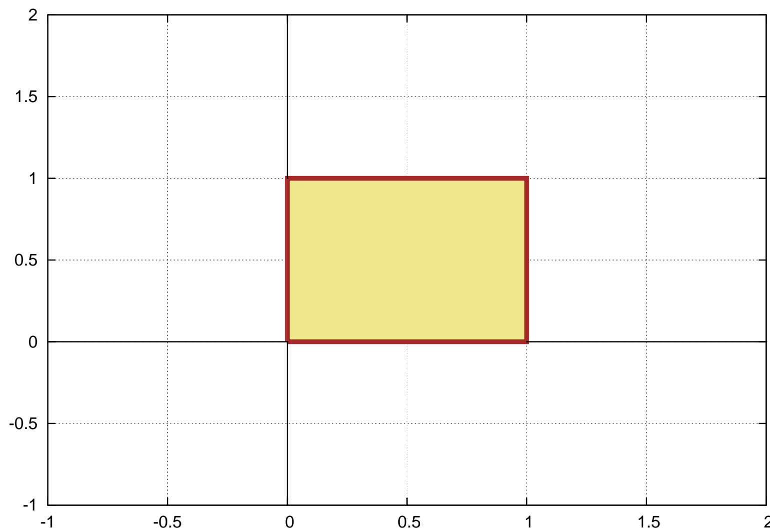

Figure 47: rect(0,0,1,1,lc(brown),lw(5),fill(khaki))

Finally, we use **rect** for a set of nested rectangles.

```
(%i15) qdraw( xr(-3,3),yr(-3,3), rect( -2.5,-2.5,2.5,2.5,lw(4),lc(blue) ),
          rect( -2,-2,2,2,lw(4),lc(red) ),
          rect( -1.5,-1.5,1.5,1.5,lw(4),lc(green) ),
          rect( -1,-1,1,1,lw(4),lc(brown) ),
          rect( -.5,-.5,.5,.5,lw(4),lc(magenta) ),
            cut(all) )$
```

which produces:

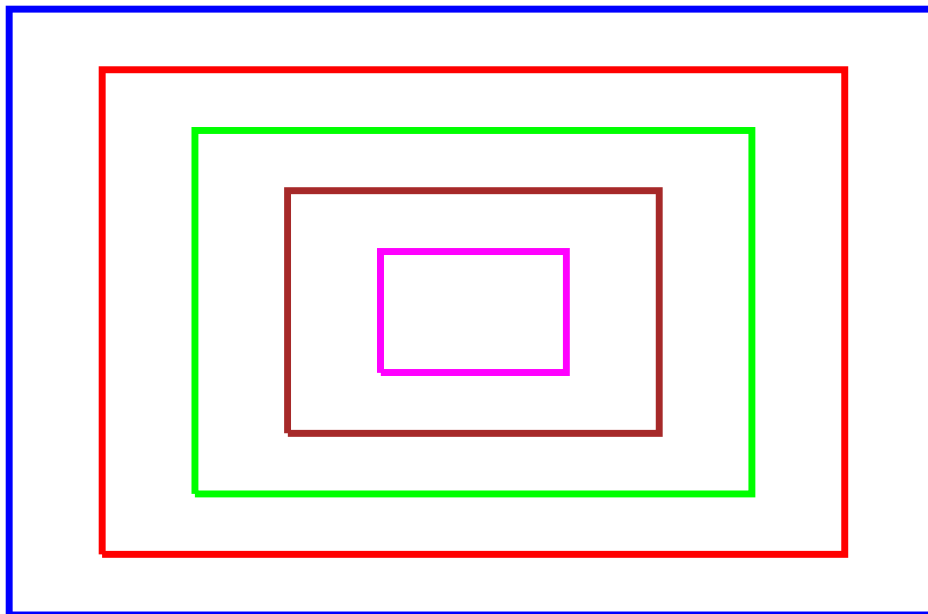

Figure 48: Nested Rectangles using rect(..)

5.13 Geometric Figures: poly(...)

The **qdraw** function **poly** has the syntax:

```
   poly( pointlist, lc(c), lw(n), fill(c) )
```

in which "pointlist" has the same form as when used with **pts**:

```
  [ [x1,y1], [x2,y2], ... [xn,yn] ] ,
```

and the arguments lc, lw, and fill are optional and can be in any order after pointlist. The last point in the list will be automatically connectd to the first.

The default call to **poly** has color black, line width 3 and no fill color.

```
(%i16) qdraw( xr(-2,2),yr(-1,2),cut(all),
             poly([ [-1,-1],[1,-1], [2,2] ]  ) )$
```

This default use of **poly** produces a ”plain jane” triangle:

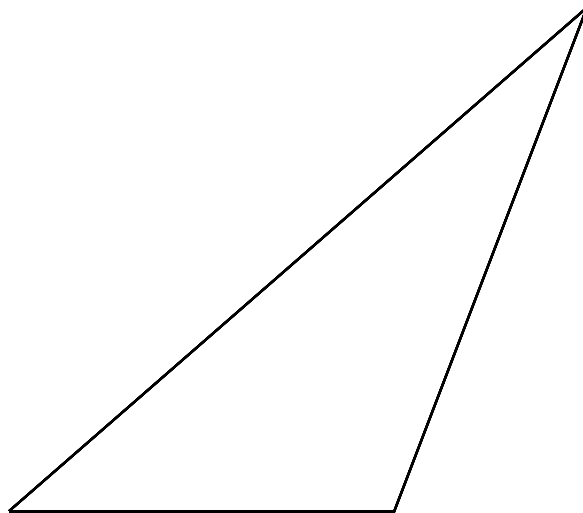

Figure 49: Default use of poly(...)

Next we create the work file ”doplot2.mac” which contains the following Maxima function which will draw eighteen right triangles in various colors:

```
/*  eighteen triangles  */
disp("doplot2()")$
print("eighteen colored triangles")$
doplot2() :=
  block([cc, qlist,x1,x2,y1,y2,i,val ],
     cc : [aquamarine,beige,blue,brown,cyan,gold,goldenrod,green,khaki,
            magenta,orange,pink,plum,purple,red,salmon,skyblue,turquoise,
            violet,yellow ],
     qlist : [ xr(-3.3,3.3), yr(-3.3,3.3) ],
  /* top row of triangles  */
     y1  : 1,
     y2 : 3,
     for i:0 thru 5 do (
       x1 : -3 + i,
       x2 : x1 + 1,
       val : poly( [ [x1,y1],[x2,y1],[x1,y2]], fill( cc[i+1] ) ),
       qlist : append(qlist, [val ] )
     ),
  /* middle row of triangles  */
     y1 : -1,
     y2 : 1,
     for i:0 thru 5 do (
        x1 : -3 + i,
        x2 : x1 + 1,
        val : poly( [ [x1,y1],[x1,y2],[x2,y2]], fill( cc[i+7] ) ),
        qlist : append(qlist, [val ] )
     ),
```

```
 /* bottom row of triangles  */
     y1 : -3,
     y2 : -1,
     for i:0 thru 5 do (
       x1 : -3 + i,
       x2 : x1 + 1,
       val : poly( [ [x1,y1],[x2,y1],[x1,y2]], fill( cc[i+13] ) ),
       qlist : append(qlist, [val ] )
     ),
     qlist : append(qlist,[ cut(all) ] ),
     apply( 'qdraw, qlist )
  )$
```

Here is a record of use of this work file:

```
(%i17) load(doplot2);
                                           doplot2()
eighteen colored triangles
(%o17)                              c:/work2/doplot2.mac
(%i18) doplot2();
```

and the resulting figure:

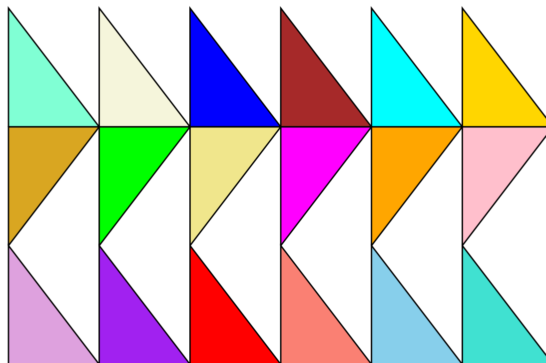

Figure 50: Using poly(...) with Color

For "homework", use **poly** and **pts** to draw the following figure. (Hint: you should also use **xr(...)**).

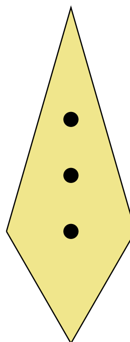

Figure 51: Homework Problem

5.14 Geometric Figures: circle(...) and ellipse(...)

The **qdraw** function **circle** has the syntax:

```
circle( xc,yc, r, lc(c), lw(n), fill(c) )
```

to draw a circle centered at `(xc,yc)` and having radius `r`. The last three arguments are optional and may be entered in any order after the required first three arguments. This object will not "look" like a circle unless you take care to make the horizontal extent of the "canvas" about 1.4 times the vertical extent (by using **xr(...)** and **yr(...)**).

Here is the default circle in black, with line width 3, and no fill color.

```
(%i19) qdraw( xr(-2, 2), yr(-2, 2), circle( 0, 0, 1 ) )$
```

which looks like:

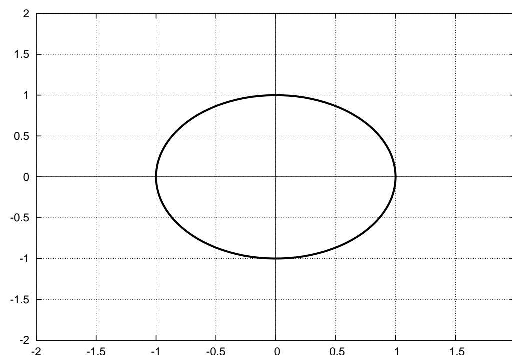

Figure 52: Default "circle"

By using **xr(...)** and **yr(...)** we try for a "round" circle and also add what should be a 45 degree line.

```
(%i20) qdraw(xr(-2.1,2.1),yr(-1.5,1.5),cut(all),
             circle(0,0,1,lw(5),lc(brown),fill(khaki) ),
               line(-1.5,-1.5,1.5,1.5,lw(8), lc(red) ),
                 key(bottom) )$
```

with the result:

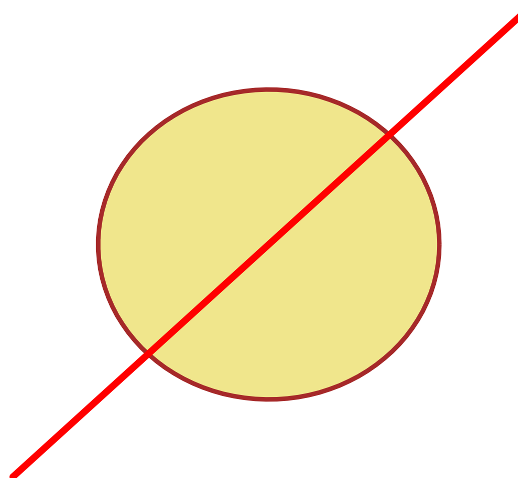

Figure 53: line over "round" circle

The line painted over the circle because of the order of the arguments to **qdraw**. If we reverse the order, drawing the line before the circle:

```
(%i21) qdraw(xr(-2.1,2.1),yr(-1.5,1.5),cut(all),
            line(-1.5,-1.5,1.5,1.5,lw(8),lc(red) ),
            circle(0,0,1,lw(8),lc(brown),fill(khaki) ),
                key(bottom) )$
```

then the circle color will hide the line:

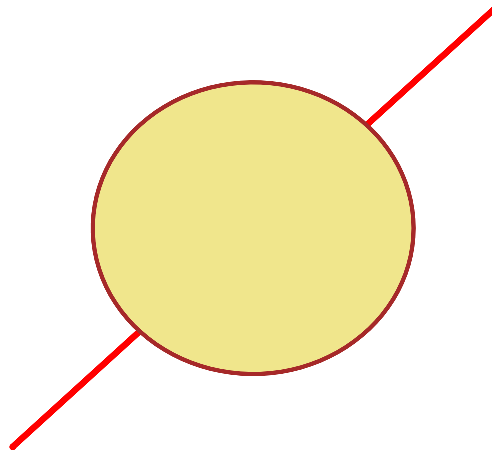

Figure 54: circle over line

The **qdraw** function **ellipse** has the syntax:
`ellipse( xc,yc,xsma,ysma,th0deg,dthdeg, lw(n), lc(c), fill(c) )`
which will plot a partial or whole ellipse centered at `(xc,yc)`, oriented with ellipse axes aligned along the `x` and `y` axes, having horizontal semi-axis xsma, vertical semi-axis ysma, beginning at th0deg degrees measured counter clockwise from the positive `x` axis, and drawn for dthdeg degrees in the counter clockwise direction.

The last three arguments are optional. The default is the outline of an ellipse for the specified angular range in color black, line width 3, and no fill color.

Here is the default behavior:

```
(%i22) qdraw( xr(-4.2,4.2),yr(-3,3),
         ellipse(0,0,3,2,90,270) )$
```

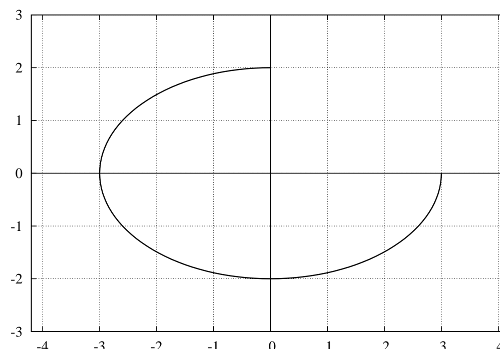

Figure 55: ellipse(0,0,3,2,90,270)

If we add color, fill, and some curvy background, as in

```
(%i23) qdraw( xr(-5.6,5.6),yr(-4,4),ex1(x,x,-4,4,lc(blue),lw(5)),
                 ex1(4*cos(x),x,-4,4,lc(red),lw(5) ),
       ellipse(0,0,3,1,90,270,lc(brown),lw(5),fill(khaki)) ,
           cut(all) )$
```

we get

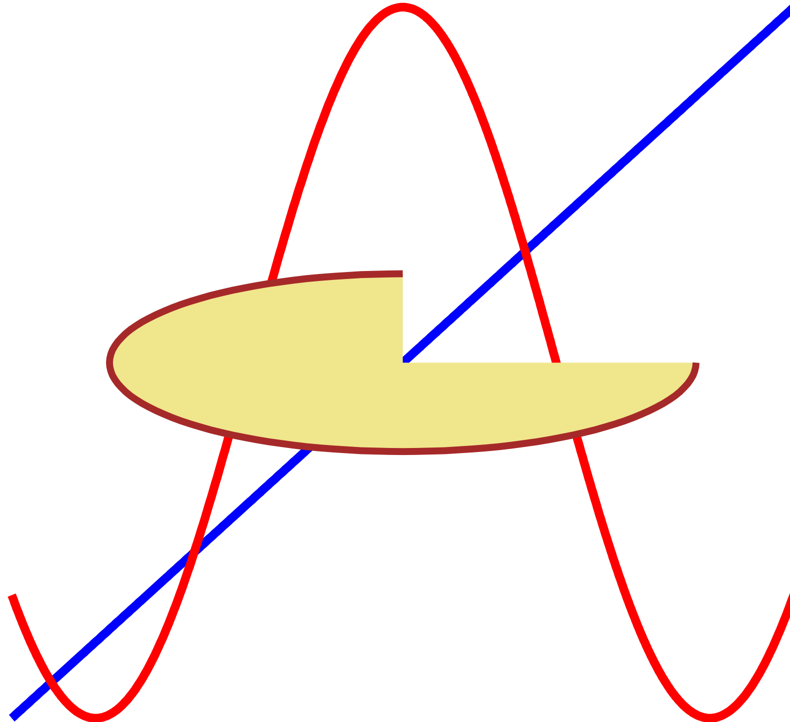

Figure 56: Filled Ellipse

5.15 Geometric Figures: vector(..)

The **qdraw** function **vector** has the syntax:
`vector([x,y],[dx,dy],ha(thdeg),hb(v),hl(v),ht(t),lw(n),lc(c),lk(string) )`
which draws a vector with components [dx,dy] starting at [x,y].
The first two list arguments are required, all others are optional and can be entered in any order after the first two required arguments.
The default ”head angle” is 30 deg; change to 45 deg using ha(45) for example.
The default ”head both” value is f for false; use hb(t) to set it to true, and hb(f) to return to false.
The default ”head length” is 0.5; use hl(0.7) to change to 0.7.
The default ”head type” is ”nofilled”; use ht(e) for ”empty”, ht(f) for ”filled”, and ht(n) to change back to ”nofilled”.
Once one of the ”head properties” has been changed in a call to **vector**, the next calls to vector assume the change is still in force.
The default line width is 3; use lw(5) to change to 5.
The default line color is black; use lc(brown) to change to brown.
The default is no key string; use lk(”A1”), for example, to create a text string for the key.

Here is a use of **vector** which accepts all defaults:

```
(%i24) qdraw( xr(-2,2), yr(-2,2), vector( [-1,-1], [2,2] ) )$
```

with the result:

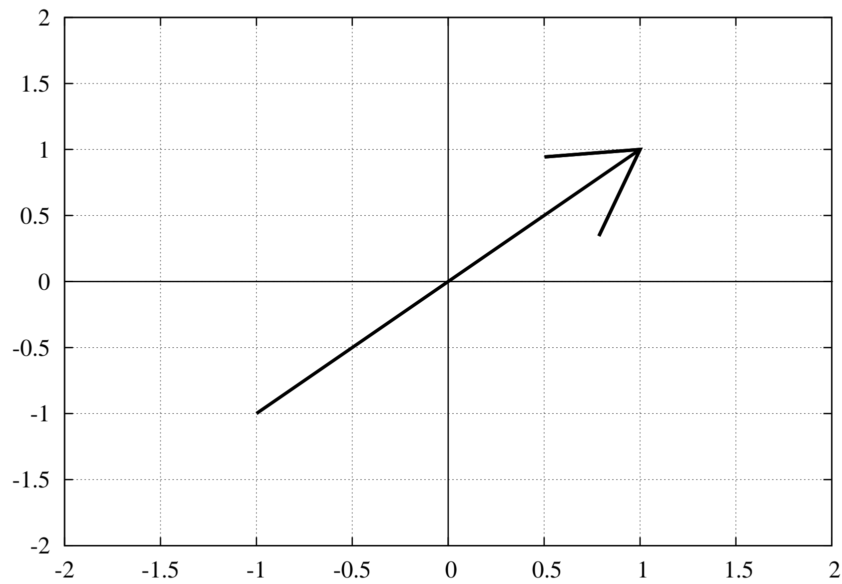

Figure 57: Default Vector

We can thicken and apply color to this basic vector with

```
(%i25) qdraw(xr(-2,2),yr(-2,2),
           vector([-1,-1],[2,2],lw(5),lc(brown),lk("vec 1")),
            key(bottom) )$
```

which looks like:

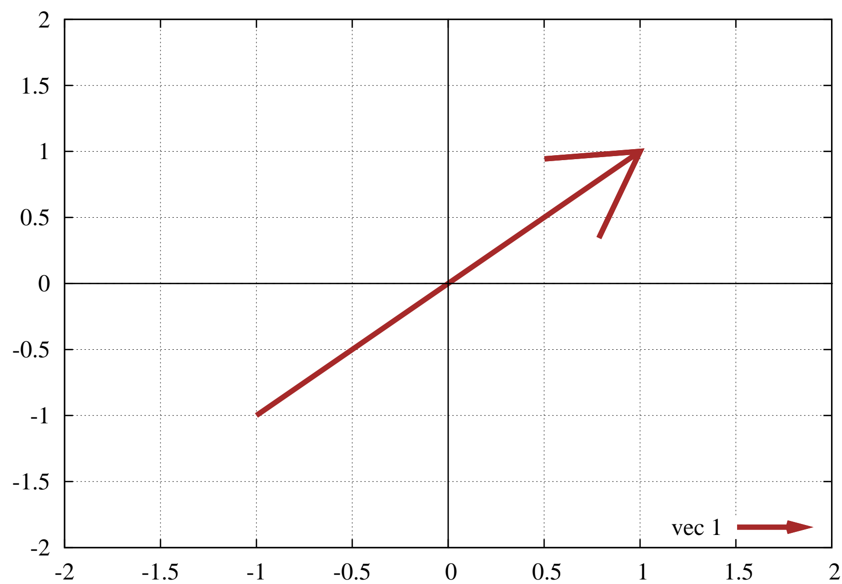

Figure 58: Adding Color, etc.

Next we can alter the "head" properties, but let's also make this vector shorter. We use `ht(e)` to set head_type to "empty", `hb(t)` to set head_both to "true", and `ha(45)` to set head_angle to 45 degrees.

```
(%i26) qdraw(xr(-2,2),yr(-2,2),
            vector([0,0],[1,1],lw(5),lc(blue),lk("vec 1"),
                    ht(e),hb(t),ha(45) ), key(bottom) )$
```

which produces:

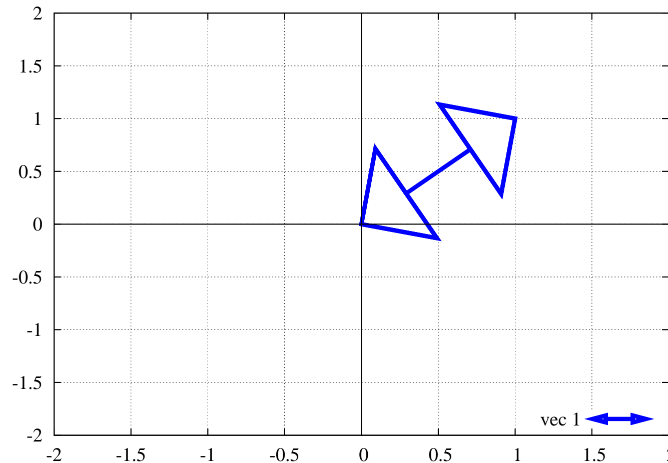

Figure 59: Changing Head Properties

Once you invoke the head properties options, the new settings are used on your next calls to **vector** (unless you deliberately change them). Here is an example of that memory feature at work:

```
(%i27) qdraw(xr(-2.8,2.8),yr(-2,2),
            vector([0,0],[1,1],lw(5),lc(blue),lk("vec 1"),
                          ht(e),hb(t),ha(45) ),
            vector([0,0],[-1,-1],lw(5),lc(red),lk("vec 2")),
             key(bottom) )$
```

and we also used the x-range setting to get the geometry closer to reality, with the result:

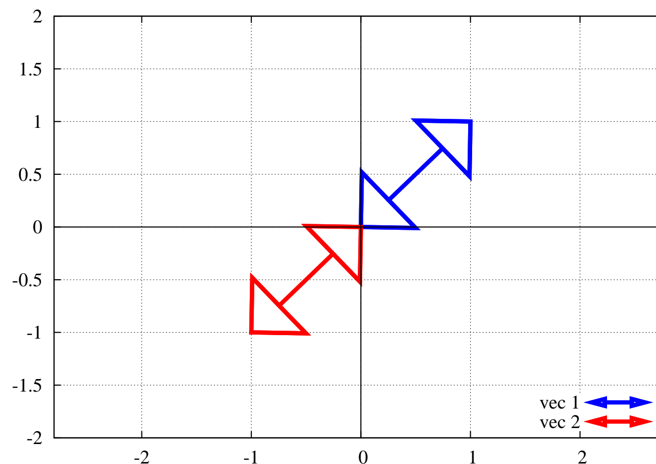

Figure 60: Head Properties Memory at Work

5.16 Geometric Figures: arrowhead(..)

The syntax of the **qdraw** function **arrowhead** is:

```
arrowhead( x, y, theta-degrees, s, lc(c), lw(n) )
```

which will draw an arrow head with the vertex at `(x,y)`.
The first four arguments are required and must be numbers.
The third argument theta is an angle in degrees and is the direction the arrowhead is to point relative to the positive x axis, ccw from x axis taken as a positive angle.
The fourth argument s is the length of the sides of the arrowhead.
The arguments lc(c) and lw(n) are optional, and are used to alter the default color (black) and line width (3).
The opening half angle is hardwired to be phi = 25 deg = 0.44 radians.
The geometry will look better if the x-range is about 1.4 times the y-range.

Here are four arrow heads drawn with the default line widths and color and "size" 0.3, which show the use of the direction argument in degrees.

```
(%i28) qdraw(xr(-2.8,2.8),yr(-2,2),
             arrowhead(1.5,0,180,.3),arrowhead(0,1,270,.3),
             arrowhead(-1.5,0,0,.3),arrowhead(0,-1,90,.3) )$
```

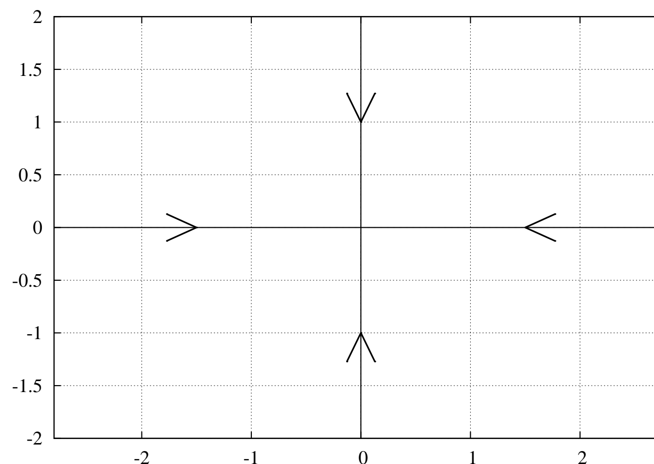

Figure 61: Using arrowhead(..)

5.17 Labels with Greek Letters

5.17.1 Enhanced Postscript Methods

Here we combine **line(..)**, **ellipse**, **arrowhead** and **label**, using the enhanced postscript abilities of **draw2d**'s `terminal = eps` option, which became effective with the June 11, 2008 draw package update. This update includes an extension of the `terminal = eps` abilities to include local conversion of font properties and the use of Greek and some math characters. As of the writing of this section, it was necessary to download this update from the webpage
`http://maxima.cvs.sourceforge.net/maxima/maxima/share/draw/draw.lisp`. On that webpage you will see `Log of /maxima/share/draw/draw.lisp`, and the top entry is
`Revision 1.31 - (view) (download) (annotate) - [select for diffs]`
`Wed Jun 11 18:19:55 2008 UTC`. Click on the "download" link, and the text of draw.lisp will appear in your browser with the top looking like:

```
;;;                    COPYRIGHT NOTICE
;;;
;;;  Copyright (C) 2007 Mario Rodriguez Riotorto
;;;
;;;  This program is free software; you can redistribute
;;;  it and/or modify it under the terms of the
;;;   ........................ etc
```

This is a program written in the Lisp language, and down near the bottom is a series of lines which provide for the enhanced postscript behavior:

```
      ($eps (format cmdstorage "set terminal postscript eps enhanced ~a
                               size ~acm, ~acm~%set out '~a.eps'"
                           (write-font-type) ; other alternatives are Arial, Courier
                           (get-option '$eps_width)
                           (get-option '$eps_height)
                           (get-option '$file_name)))
```

If you save this text file with the name "draw.lisp" and place it in the draw package folder (on my Windows computer, the rather complicated path:
`drive c, program files, maxima-5.15.0, share, maxima, 5.15.0, share, draw )`
to replace the old "draw.lisp" (which you could rename prior to the save as), then you can use the label syntax inside strings as shown in the following.

```
(%i29) qdraw(xr(0,2.8),yr(0,2),
             line(0,0,2.8,0,lw(2)),
             line(0,0,2,2,lc(blue),lw(8) ),
           ellipse(0,0,1,1,0,45 ),
           arrowhead(0.707,0.707,135,0.15),
   label(["{/=36 {/Symbol q   \\254 }  The Incline Angle}",1,0.4]),
            cut(all),
           pic(eps,"ch5p52" ) );
```

The result looks like:

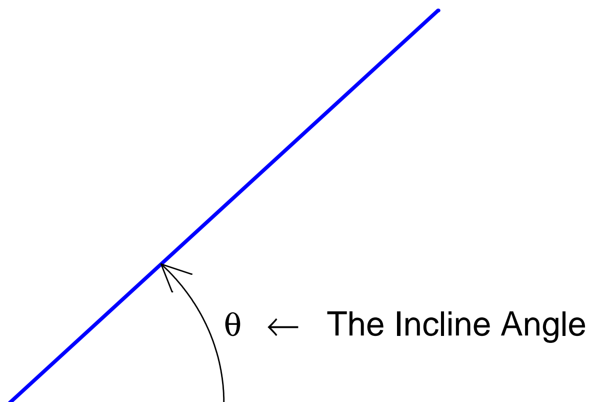

Figure 62: line(..), ellipse(..), arrowhead(..), label(..)

A summary of the enhanced postscript syntax can be downloaded:
`http://www.telefonica.net/web2/biomates/maxima/gpdraw/ps/ps_guide.ps` although the examples of using the "characters" works for me only if I use two back slashes, as in the example just shown,

where the entry `\\254` inside the `{/Symbol }` structure produces the leftward pointing arrow (thanks to the draw package developer, Mario Rodriguez Riotorto, for the enhanced postscript abilities, and for aiding my understanding of how to use these features). An example of creating text for a label which includes the integral sign and Greek letters is given on the webpage:
`http://www.telefonica.net/web2/biomates/maxima/gpdraw/ps/`.

The entry `{/Symbol q }` by itself would produce just the Greek letter θ. Wrapping the text entry in the structure `{/=36 }` accepts the default font type and sets the font size to 36 for the text inside the pair of braces.

Here we make a lower case Latin to Greek conversion reminder using four instances of **label**, although we could alternatively have used the syntax `label( [s1,x1,x2], [s2,x2,y2],... )`.

```
(%i30) qdraw(xr(-3,3),yr(-2,2),label_align(c),
         label( ["{/=48 a b c d e f g h i j k l m}",0,1.5] ),
        label( ["{/Symbol=48 a b c d e f g h i j k l m}",0,0.5] ),
        label( ["{/=48 n o p q r s t u v w x y z}",0,-.5] ),
         label( ["{/Symbol=48 n o p q r s t u v w x y z}",0,-1.5] ),
        cut(all), pic(eps,"ch5p53") )$
```

Note how we increase the font size of the latin alphabet `a,b,c...`. Here is the resulting eps figure:

a b c d e f g h i j k l m

α β χ δ ε ϕ γ η ι φ κ λ μ

n o p q r s t u v w x y z

ν ο π θ ρ σ τ υ ϖ ω ξ ψ ζ

Figure 63: Lower Case Latin to Greek

We can repeat that label figure using upper case Latin letters:

```
(%i31) qdraw(xr(-3,3),yr(-2,2),label_align(c),
         label( ["{/=48 A B C D E F G H I J K L M}",0,1.5] ),
        label( ["{/Symbol=48 A B C D E F G H I J K L M}",0,0.5] ),
        label( ["{/=48 N O P Q R S T U V W X Y Z}",0,-.5] ),
         label( ["{/Symbol=48 N O P Q R S T U V W X Y Z}",0,-1.5] ),
        cut(all), pic(eps,"ch5p54") )$
```

You can see the resulting figure on the next page.

Useful character codes, used as `{/Symbol \\abc \\rst etc }`
or as `{/Symbol=36 \\abc \\rst etc }` are:
`\\243` (less than or equal)
`\\245` (infinity symbol)
`\\253` (double ended arrow)
`\\254` (left arrow)
`\\256` (right arrow)

\\261 (plus or minus)
\\263 (greater than or equal)
\\264 (times)
\\271 (not equal)
\\273 (approx equal)
\\345 (summation sign)
\\362 (integral sign)

A B C D E F G H I J K L M

Α Β Χ Δ Ε Φ Γ Η Ι ϑ Κ Λ Μ

N O P Q R S T U V W X Y Z

Ν Ο Π Θ Ρ Σ Τ Υ ς Ω Ξ Ψ Ζ

Figure 64: Upper Case Latin to Greek

Here we use **label** to illustrate these possible symbols you can use:

```
(%i32) s1 : "{/Symbol=48 \\243   \\245   \\253   \\254   \\256}"$
(%i33) s2 : "{/Symbol=48 \\261   \\263   \\264   \\271   \\273   \\345   \\362}"$
(%i34) qdraw(xr(-3,3),yr(-2,2),label_align(c),
        label( [s1,0,1] ), label( [s2,0,-1] ), cut(all), pic(eps,"ch5p55") )$
```

which produces the figure:

$\leq \quad \infty \quad \leftrightarrow \quad \leftarrow \quad \rightarrow$

$\pm \quad \geq \quad \times \quad \neq \quad \approx \quad \sum \quad \int$

Figure 65: Useful Character Code Symbols

5.17.2 Windows Fonts Methods with jpeg Files

We can produce the Greek letter θ in a jpeg file or a png file by using the Greek font files in the `c:\windows\fonts` folder as follows:

```
(%i35) qdraw(xr(0,2.8),yr(0,2),
             line(0,0,2.8,0),
             line(0,0,2,2,lc(blue),lw(5) ),
           ellipse(0,0,1,1,0,45 ),
           arrowhead(0.707,0.707,135,0.15),
            label(["q",1,0.4]), cut(all),
           pic(jpg,"ch5p52p",font("c:/windows/fonts/grii.ttf",36)) );
```

which produces the file `ch5p52.jpg`, which we converted to an eps file using cygwin's convert function: `convert name.jpg name.eps` which will also work to convert a png graphics file.

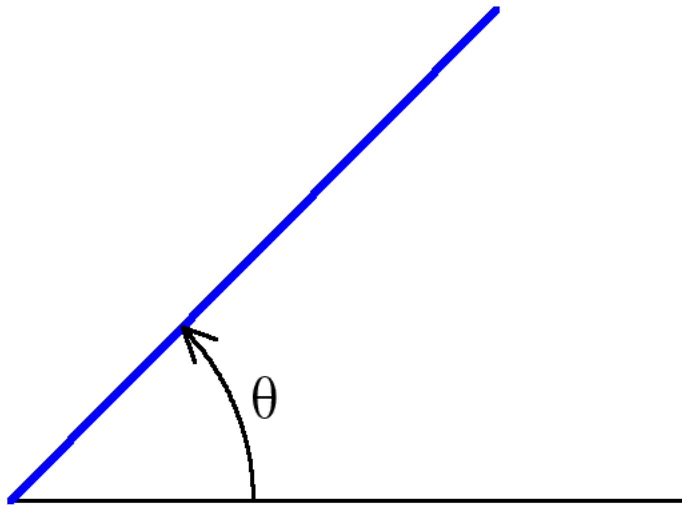

Figure 66: Greek in jpg converted to eps

Here we use the label function to show all the greek letters available via the windows, fonts folder:

```
(%i36) qdraw(xr(-3,3),yr(-2,2),
     label(["a b c d e f g h i j k l m",-2.5,1.5],
             ["n o p q r s t u v w x y z",-2.5,0.5],
              ["A B C D E F G H I J K L M",-2.5,-0.5],
                ["N O P Q R S T U V W X Y Z",-2.5,-1.5]  ),
            cut(all),
           pic(png,"ch5p60",font("c:/windows/fonts/grii.ttf",24)) );
```

After conversion of the png to an eps graphics file using Cygwin's convert function, we get:

α β χ δ ϵ ϕ γ η ι' κ λ μ

ν ο π θ ρ σ τ υ' ω ξ ψ ζ

Α Β Χ Δ Ε Φ Γ Η Ι' Κ Λ Μ

Ν Ο Π Θ Ρ Σ Τ ϒ ' Ω Ξ Ψ Ζ

Figure 67: windows fonts conv. of a - Z to Greek

5.17.3 Using Windows Fonts with the Gnuplot Console Window

In the default Windows Gnuplot console mode, you can convert some Latin letters to Greek as follows:

```
(%i37) qdraw(xr(0,2.8),yr(0,2),
            line(0,0,2.8,0),
            line(0,0,2,2,lc(blue),lw(5) ),
          ellipse(0,0,1,1,0,45 ),
          arrowhead(0.707,0.707,135,0.15),
           label(["q",1,0.4]), cut(all) );
```

When the console graphics window appears, right click on the upper left corner icon and select Options, Choose Font. In the Font panels, choose Graecall font, "regular" from the middle panel, and size 36 from the right panel and click "ok". The English letter "q" (lower case) is then converted to the Greek lower case theta. Use the Gnuplot window menu again to save the resulting image to the clipboard, and open an image viewer. I use the freely available Infanview. If you use View, Paste, the clipboard image appears inside Infanview, and you can save the image as a jpeg file in your choice of folder. Since I am using eps graphics files for this latex file, I converted the jpg to eps using Cygwin's convert function:
`convert name.jpg  name.eps.`

Here is the result:

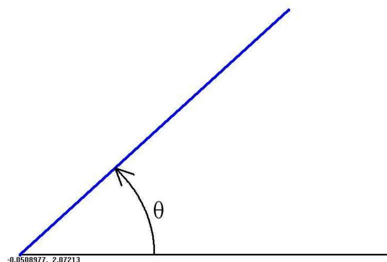

Figure 68: Greek via Windows Clipboard

Unfortunately, saving the Gnuplot window image to the Window's clipboard also saves the current cursor position, which is not desirable.

5.18 Even More with more(...)

You can use the **qdraw** function **more(...)**, containing any legal **draw2d** elements, as we illustrate by adding a label to the x-axis and a title. We focus here on producing an eps graphics file to display the enhanced ability to show subscripts and superscripts.

```
(%i38)  qdraw( lw(8), ex([x,x^2,x^3],x,-2,2),
          more(xlabel = "X AXIS", title="intersections of x, x^2, x^3" ),
   cut(key),line(-2,0,2,0,lw(2)),line(0,-8,0,8,lw(2)),
               vector([-1,5],[-0.4,-2.7],lc(red),hl(0.1) ),
               label(["x^2",-0.9,6]),
               vector([-1.2,-6],[-0.5,0],lc(turquoise),lw(8)),
               label( ["x^3", -1,-5.5] ),
              pts( [[-1,-1],[0,0],[1,1]],ps(2),pc(magenta)  ),
              pic(eps,"ch5p56",font("Times-Roman",28)) )$
```

The lines for the x and y axes need special emphasis to show up clearly with an eps file, so we have used **qdraw**'s **line** function for that task. We also need to increase the line width setting for the eps file case, which we have done with the **qdraw** top level function **lw**, which only affects the "quick plotting" functions **ex** and **imp**. We have used **qdraw**'s **more** function to provide an x-axis label and a title. The font setting in the **pic** function supplies an overall drawing font type and size which affects all elements unless locally over-ridden with the special enhanced postscript features. In the title and labels, `x^n` is converted automatically to x^n.

Here is the resulting plot:

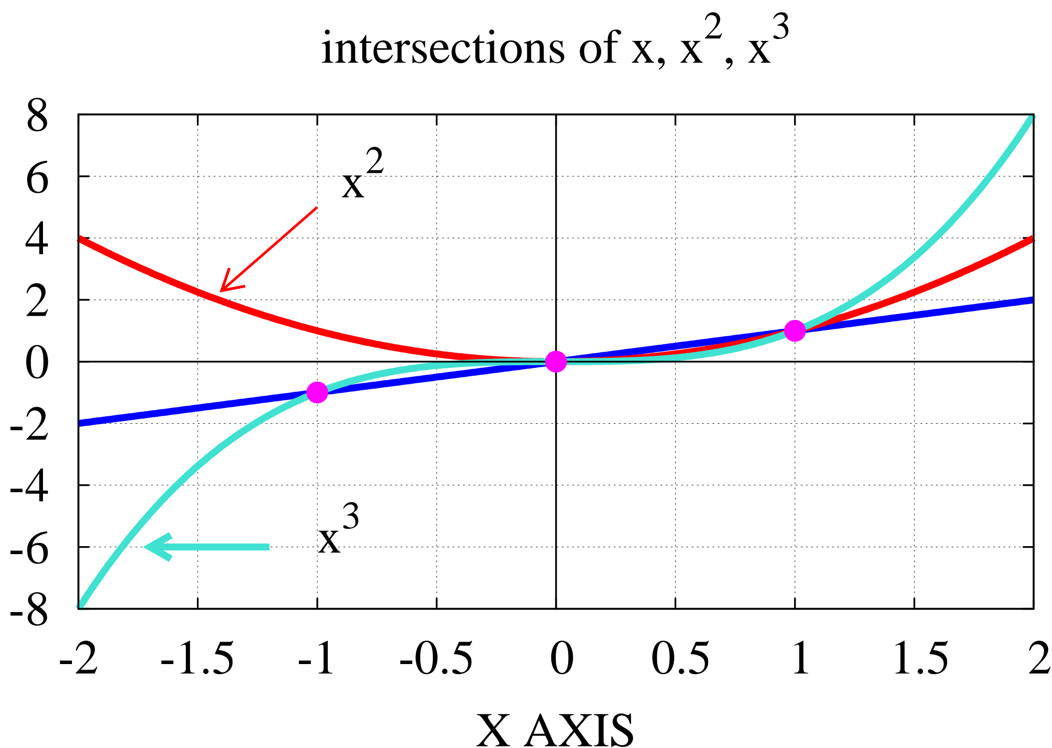

Figure 69: Using more(...) for Title and X Axis Label

5.19 Programming Homework Exercises

5.19.1 General Comments

The file qdraw.mac is a text file which you can modify with a good text editor such as notepad2. This Maxima code is heavily commented as an aid to passing on some Maxima language programming examples. You can get some experience with the Maxima programming language elements by copying the file qdraw.mac to another name, say myqdraw.mac, and use that copy to make modifications to the code which might interest you. By frequently loading in the modified file with `load(myqdraw)`, you can let Maxima check for syntax errors, which it does immediately.

The most common syntax errors involve parentheses and commas, with strange error messages such as " BLANK IS NOT AN INFIX OPERATOR", or "TOO MANY PARENTHESES", etc. Placing a comma just before a closing parenthsis is a fatal error which can nevertheless creep in. This sounds obvious, but you may find it useful to insert some special debug printouts, such as `print("in blank, a = ",a)` or `display(a)`, perhaps at the end of a do loop, so you are working with the structure:

```
  for i thru n do (
    job1,
    job2,
    job3,
print(" i = ",i,"  blank = ",blank)
  /* end do loop */
),
...program continues...
```

When you are finished debugging a section, you either will comment out the debug printout or simply delete it to clean up the code. If you are not fully awake, you might then load into Maxima

```
  for i thru n do (
    job1,
    job2,
    job3,
      /* end do loop */
),
...program continues...
```

and, of course, Maxima will object, since that extra comma no longer makes sense.

It is crucial to use a good text editor which will "balance" parentheses, brackets, and braces to minimize parentheses etc errors.

If you look at the general structure of **qdraw**, you will see that most of the real work is done by **qdraw1**. If you call **qdraw1** instead of **qdraw**, you will be presented with a rather long list of elements which are understood by **draw2d**. Even if you use **qdraw**, you will see the same long list wrapped by "draw2d" if you have not loaded the **draw** package.

One feature you should look at is how a function which takes an arbitrary number of arguments, depending on the user (as does the function **draw2d**), is defined. If this seems strange to you, experiment with a toy function having a variable number of arguments, and use printouts inside the function to see what Maxima is doing.

5.19.2 XMaxima Tips

It is useful to first try out a small code idea directly in XMaxima, even if the code is ten or fifteen lines long, since the XMaxima interface has been greatly improved. When you want to edit your previous "try", use `Alt-p` to enter your previous code, and immediately backspace over the final `);` or `)$`. You can then cursor up to an area where you want to add a new line of code, and with the cursor placed just after a comma, press `ENTER` to create a new (blank) line. Since the block of code has not been properly concluded with either a `);` or `)$`, Maxima will not try to "run" with the version you are working on when you press `ENTER`. Once you have made the changes you want, cursor your way to the end and put back the correct ending and then pressing `ENTER` will send your code to the Maxima engine.

The use of `HOME`, `END`, `PAGEUP`, `PAGEDOWN`, `CNTRL-HOME`, and `CNTRL-END` greatly speeds up working with XMaxima. For example to copy a code entry up near the top of your current workspace, first enter `HOME` to put the cursor at the beginning of the current line, then `PAGEUP` or `CNTRL-HOME` to get up to the top fast, then drag over the code (don't include the `(%i5)` part) to the end but not to the concluding `);` or `)$`. You can hold down the `SHIFT` key and use the right (and left) cursor key to help you select a region to copy. Then press `CNTRL-C` to copy the selected code to Window's clipboard. Then press `CTRL-END` to have the cursor move to the bottom of your workspace where XMaxima is waiting for your next input. Press `CNTRL-V` to paste your selection. If the selection extends over multiple lines, use the down cursor key to find the end of the selection which should be without the proper code ending `);` or `)$`. You are then in the driver's seat and can cursor your way around the code and make any changes without danger of XMaxima pre-emptively sending your work to the computing engine until you go to that end and provide the proper ending.

5.19.3 Suggested Projects

You will have noticed that we used the **qdraw** function **more** in order to insert axis labels and a title into our plot. Design **qdraw** functions `xlabel(string)`, `ylabel(string)`, and `title(string)`. Place them in the "scan 3" section of **qdraw** and try them out. You will need to pay attention to how new elements get passed to **draw2d**. In particular, look at the list `drlist`, using your text editor search function (in notepad2, Ctrl-f) to see how that list is constructed based on the user input.

A second small project would be to add a "line type" option for the **qdraw** function **line**. My experience is that setting `line_type = dots` in **draw2d** produces no immediate change in the Windows Gnuplot console window, produces a finely dotted line for jpeg and png image files, and produces a dashed line with eps image files. Your addition to **qdraw** should follow the present style, so the user would use the syntax `line(x1,y1,x2,y2,lc(c),lw(n),lk(string),lt(type) )`, where type is either s or d (for solid or dots).

A third small project would be to design a function **triangle** for **qdraw**, including the options which are presently in **poly**.

A fourth small project would be to include the option cbox(..) in the **qdensity** function. The present default is to include the colorbox key next to the density plot, but if the user entered `qdensity(....,cbox(f) )` , the colorbox would be removed.

A more challenging project would be to write a **qdraw** function which would directly access the creation of bar charts. These notes are written with the needs of the typical physical science or engineering user in mind, so no attention has been paid to bar charts here. Naturally, if you frequently construct bar charts, this project would be interesting for you. Start this project by first working with **draw2d** directly, to get familiar with what is already available, and to avoid "re-creating the wheel".

One general principle to keep in mind is that the Maxima language is an "interpretive language"; Maxima does not make multiple passes over your code to reconcile function references, such as a compiler does. This means that if a part of your code "calls" some user defined function, Maxima needs to have already read about that function definition in your code.

5.20 Acknowledgements

The author would like to thank Mario Rodriguez Riotorto, the creator of Maxima's **draw** graphics interface to Gnuplot, for his encouragement and advice at crucial stages in the development of the **qdraw** interface to **draw2d**. The serious graphics user should spend time with the many powerful features of the **draw** package, and the examples provided on the **draw** webpages,
`http://www.telefonica.net/web2/biomates/maxima/gpdraw/`.
These examples go far beyond the simple graphics in this chapter. The recent updating of the **draw** package to allow use of Gnuplot's enhanced postscript features makes Maxima a more attractive tool for the creation of educational diagrams.

Maxima by Example: Ch.6: Differential Calculus *

Edwin L. Woollett

October 21, 2010

Contents

6 Differential Calculus **3**
6.1 Differentiation of Explicit Functions . . . 4
6.1.1 All About **diff** . . . 4
6.1.2 The Total Differential . . . 5
6.1.3 Controlling the Form of a Derivative with **gradef** . . . 6
6.2 Critical and Inflection Points of a Curve Defined by an Explicit Function . . . 7
6.2.1 Example 1: A Polynomial . . . 7
6.2.2 Automating Derivative Plots with **plotderiv** . . . 9
6.2.3 Example 2: Another Polynomial . . . 11
6.2.4 Example 3: $\mathbf{x^{2/3}}$, Singular Derivative, Use of **limit** . . . 12
6.3 Tangent and Normal of a Point of a Curve Defined by an Explicit Function . . . 13
6.3.1 Example 1: $\mathbf{x^2}$ . . . 14
6.3.2 Example 2: $\mathbf{\ln(x)}$ . . . 14
6.4 Maxima and Minima of a Function of Two Variables . . . 15
6.4.1 Example 1: Minimize the Area of a Rectangular Box of Fixed Volume . . . 16
6.4.2 Example 2: Maximize the Cross Sectional Area of a Trough . . . 18
6.5 Tangent and Normal of a Point of a Curve Defined by an Implicit Function . . . 19
6.5.1 Tangent of a Point of a Curve Defined by $\mathbf{f(x,y) = 0}$ . . . 21
6.5.2 Example 1: Tangent and Normal of a Point of a Circle . . . 23
6.5.3 Example 2: Tangent and Normal of a Point of the Curve $\mathbf{\sin(2\,x)\ \cos(y) = 0.5}$ . . . 25
6.5.4 Example 3: Tangent and Normal of a Point on a Parametric Curve: $\mathbf{x = \sin(t), y = \sin(2\,t)}$ . . . 26
6.5.5 Example 4: Tangent and Normal of a Point on a Polar Plot: $\mathbf{x = r(t)\cos(t), y = r(t)\sin(t)}$ . . . 27
6.6 Limit Examples Using Maxima's **limit** Function . . . 28
6.6.1 Discontinuous Functions . . . 29
6.6.2 Indefinite Limits . . . 32
6.7 Taylor Series Expansions using **taylor** . . . 34
6.8 Vector Calculus Calculations and Derivations using **vcalc.mac** . . . 36
6.9 Maxima Derivation of Vector Calculus Formulas in **Cylindrical Coordinates** . . . 40
6.9.1 The Calculus Chain Rule in Maxima . . . 41
6.9.2 Laplacian $\nabla^2 f(\rho, \varphi, z)$ . . . 43
6.9.3 Gradient $\nabla f(\rho, \varphi, z)$ . . . 45
6.9.4 Divergence $\nabla \cdot \mathbf{B}(\rho, \varphi, z)$ . . . 48
6.9.5 Curl $\nabla \times \mathbf{B}(\rho, \varphi, z)$ . . . 49
6.10 Maxima Derivation of Vector Calculus Formulas in **Spherical Polar Coordinates** . . . 50

*This version uses Maxima 5.21.1 This is a live document. Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions to `woollett@charter.net`

Preface

```
COPYING AND DISTRIBUTION POLICY
This document is part of a series of notes titled
"Maxima by Example" and is made available
via the author's webpage http://www.csulb.edu/~woollett/
to aid new users of the Maxima computer algebra system.

NON-PROFIT PRINTING AND DISTRIBUTION IS PERMITTED.

You may make copies of this document and distribute them
to others as long as you charge no more than the costs of printing.

These notes (with some modifications) will be published in book form
eventually via Lulu.com in an arrangement which will continue
to allow unlimited free download of the pdf files as well as the option
of ordering a low cost paperbound version of these notes.
```

Feedback from readers is the best way for this series of notes to become more helpful to new users of Maxima. *All* comments and suggestions for improvements will be appreciated and carefully considered.

```
LOADING FILES
The defaults allow you to use the brief version load(fft) to load in the
Maxima file fft.lisp.
To load in your own file, such as qxxx.mac
using the brief version load(qxxx), you either need to place
qxxx.mac in one of the folders Maxima searches by default, or
else put a line like:

file_search_maxima : append(["c:/work2/###.{mac,mc}"],file_search_maxima )$

in your personal startup file maxima-init.mac (see later in this chapter
for more information about this).

Otherwise you need to provide a complete path in double quotes,
as in load("c:/work2/qxxx.mac"),

We always use the brief load version in our examples, which are generated
using the XMaxima graphics interface on a Windows XP computer, and copied
into a fancy verbatim environment in a latex file which uses the fancyvrb
and color packages.
```

```
Maxima.sourceforge.net. Maxima, a Computer Algebra System. Version 5.21.1
 (2010). http://maxima.sourceforge.net/
```

The homemade function **fll(x)** (first, last, length) is used to return the first and last elements of lists (as well as the length), and is automatically loaded in with **mbe1util.mac** from Ch. 1. We will include a reference to this definition when working with lists.

This function has the definitions

```
fll(x) := [first(x),last(x),length(x)]$
declare(fll,evfun)$
```

Some of the examples used in these notes are from the Maxima html help manual or the Maxima mailing list:
http://maxima.sourceforge.net/maximalist.html.

The author would like to thank the Maxima developers for their friendly help via the Maxima mailing list.

6 Differential Calculus

The methods of calculus lie at the heart of the physical sciences and engineering.

The student of calculus needs to take charge of his or her own understanding, taking the subject apart and putting it back together again in a way that makes logical sense. The best way to learn any subject is to work through a large collection of problems. The more problems you work on your own, the more you "own" the subject. Maxima can help you make faster progress, if you are just learning calculus.

For those who are already calculus savvy, the examples in this chapter will offer an opportunity to see some Maxima tools in the context of very simple examples, but you will likely be thinking about much harder problems you want to solve as you see these tools used here.

After examples of using **diff** and **gradef**, we present examples of finding the critical and inflection points of plane curves defined by an explicit function.

We then present examples of calculating and plotting the tangent and normal to a point on a curve, first for explicit functions and then for implicit functions.

We also have two examples of finding the maximum or minimum of a function of two variables.

We then present several examples of using Maxima's powerful **limit** function, followed by several examples of using **taylor**.

The next section shows examples of using the vector calculus functions (as well as cross product) in the package **vcalc.mac**, developed by the author and available on his webpage with this chapter, to calculate the gradient, divergence, curl, and Laplacian in cartesian, cylindrical, and spherical polar coordinate systems. An example of using this package would be **curl([r*cos(theta),0,0]);** (if the current coordinate system has already been changed to spherical polar) or **curl([r*cos(theta),0,0], s(r,theta,phi));** if the coordinate system needs to be shifted to spherical polar from either cartesian (the starting default) or cylindrical.

The order of list vector components corresponds to the order of the arguments in **s(r,theta,phi)**. The Maxima output is the list of the vector curl components in the current coordinate system, in this case **[0, 0, sin(theta)]** plus a reminder to the user of what the current coordinate system is and what symbols are currently being used for the independent variables.

Thus the syntax is based on lists and is similar (although better!) than Mathematica's syntax.

There is a separate function to change the current coordinate system. To set the use of cylindrical coordinates **(rho,phi,z)**:

```
setcoord( cy(rho,phi,z) );
```

and to set cylindrical coordinates **(r,t,z)**:

```
setcoord( cy(r,t,z) );
```

The package **vcalc.mac** also contains the plotting function **plotderiv** which is useful for "automating" the plotting of a function and its first **n** derivatives.

The next two sections dicuss the use of the batch file mode of problem solving by presenting Maxima based derivations of the form of the gradient, divergence, curl, and Laplacian by starting with the cartesian forms and changing variables to (separately) cylindrical and spherical polar coordinates. (The batch files used are **cylinder.mac** and **sphere.mac**.) These two sections show Maxima's implementation of the calculus chain rule at work with use of both **depends** and **gradef**.

6.1 Differentiation of Explicit Functions

We begin with explicit functions of a single variable. After giving a few examples of the use of Maxima's **diff** function, we will discuss critical and inflection points of curves defined by explicit functions, and the construction and plotting of the tangent and normal of a point of such curves.

6.1.1 All About diff

The command **diff(expr,var,num)** will differentiate the expresssion in slot one with respect to the variable entered in slot two a number of times determined by a positive integer in slot three. Unless a dependency has been established, all parameters and "variables" in the expression are treated as constants when taking the derivative.

Thus **diff(expr,x,2)** will yield the second derivative of **expr** with respect to the variable **x**.

The simple form **diff(expr, var)** is equivalent to **diff(expr,var,1)**.

If the expression depends on more than one variable, we can use commands such as **diff(expr,x,2,y,1)** to find the result of taking the second derivative with respect to **x** (holding **y** fixed) followed by the first derivative with respect to **y** (holding **x** fixed).

Here are some simple examples.

We first calculate the derivative of x^n.

```
(%i1) diff(x^n,x);
                                          n - 1
(%o1)                                  n x
```

Next we calculate the third derivative of x^n.

```
(%i2) diff(x^n,x,3);
                                                      n - 3
(%o2)                         (n - 2) (n - 1) n x
```

Here we take the derivative (with respect to x) of an expression depending on both x and y.

```
(%i3) diff(x^2 + y^2,x);
(%o3)                                     2 x
```

You can differentiate with respect to any expression that does not involve explicit mathematical operations.

```
(%i4) diff( x[1]^2 + x[2]^2, x[1] );
(%o4)                                   2 x
                                           1
```

Note that x_1 is Maxima's way of "pretty printing" **x[1]**. We can use the **grind** function to display the output **%o4** in the "non-pretty print mode" (what would have been returned if we had set the **display2d** switch to **false**).

```
(%i5) grind(%)$
2*x[1]$
(%i6) display2d$
```

Note the dollar sign **grind** adds to the end of its output.

Finally, an example of using one invocation of **diff** to differentiate with respect to more than one variable:

```
(%i7) diff(x^2*y^3,x,1,y,2);
(%o7)                                  12 x y
```

6.1.2 The Total Differential

If you use the **diff** function without a symbol in slot two, Maxima returns the "total differential" of the expression in slot one, and by default assumes every parameter is a variable.

If an expression contains a single parameter, say **x**, then **diff(expr)** will generate

$$\mathrm{d}\,\mathrm{f}(\mathrm{x}) = \left(\frac{\mathrm{d}\,\mathrm{f}(\mathrm{x})}{\mathrm{d}\,\mathrm{x}}\right)\,\mathrm{d}\,\mathrm{x} \tag{6.1}$$

In the first example below, we calculate the differential of the expression x^2, that is, the derivative of the expression (with respect to the variable x) multiplied by "the differential of the independent variable x", $\mathrm{d}\,x$. Maxima uses **del(x)** for the differential of x, a small increment of x.

```
(%i1) diff(x^2);
(%o1)                          2 x del(x)
```

If the expression contains two parameters **x** and **y**, then the total differential is equivalent to the Maxima expression

diff(expr,x)*del(x) + diff(expr,y)*del(y) ,

which generates (using conventional notation):

$$\mathrm{d}\,\mathrm{f}(\mathrm{x},\mathrm{y}) = \left(\frac{\partial \mathrm{f}(\mathrm{x},\mathrm{y})}{\partial \mathrm{x}}\right)_{\mathrm{y}}\,\mathrm{d}\,\mathrm{x} + \left(\frac{\partial \mathrm{f}(\mathrm{x},\mathrm{y})}{\partial \mathrm{y}}\right)_{\mathrm{x}}\,\mathrm{d}\,\mathrm{y}. \tag{6.2}$$

Each additional parameter induces an additional term of the same form.

```
(%i2) diff(x^2*y^3);
                          2  2                3
(%o2)                  3 x  y  del(y) + 2 x y  del(x)
(%i3) diff(a*x^2*y^3);
                   2  2                  3            2  3
(%o3)         3 a x  y  del(y) + 2 a x y  del(x) + x  y  del(a)
```

We can use the **subst** function to replace, say **del(x)**, by anything else:

```
(%i4) subst(del(x) = dx, %o1);
(%o4)                            2 dx x
(%i5) subst([del(x)= dx, del(y) = dy, del(a) = da],%o3 );
                         2  3              3          2  2
(%o5)                da x  y  + 2 a dx x y  + 3 a dy x  y
```

You can use the **declare** function to prevent some of the symbols from being treated as variables when calculating the total differential:

```
(%i6) declare (a, constant)$
(%i7) diff(a*x^2*y^3);
                          2  2                  3
(%o7)                3 a x  y  del(y) + 2 a x y  del(x)
(%i8) declare([b,c],constant)$
(%i9) diff( a*x^3 + b*x^2 + c*x );
                               2
(%o9)                    (3 a x  + 2 b x + c) del(x)
(%i10) properties(a);
(%o10)                 [database info, kind(a, constant)]
(%i11) propvars(constant);
(%o11)                            [a, b, c]
```

```
(%i12) kill(a,b,c)$
(%i13) propvars(constant);
(%o13)                                  []
(%i14) diff(a*x);
(%o14)                         a del(x) + x del(a)
```

We can "map" **diff** on to a list of functions of **x**, say, and divide by **del(x)**, to generate a list of derivatives, as in

```
(%i15) map('diff,[sin(x),cos(x),tan(x) ] )/del(x);
                                                  2
(%o15)                        [cos(x), - sin(x), sec (x)]
(%i16) map('diff,%)/del(x);
                                                2
(%o16)                  [- sin(x), - cos(x), 2 sec (x) tan(x)]
```

6.1.3 Controlling the Form of a Derivative with gradef

We can use **gradef** to select one from among a number of alternative ways of writing the result of differentiation. The large number of "trigonometric identities" means that any given expression containing trig functions can be written in terms of a different set of trig functions.

As an example, consider trig identities which express $\sin(x)$, $\cos(x)$, and $\sec^2(x)$ in terms of $\tan(x)$ and $\tan(x/2)$:

$$\sec^2(x) = 1 + \tan^2(x), \tag{6.3}$$

$$\sin x = 2\tan(x/2)/(1 + \tan^2(x/2)), \tag{6.4}$$

$$\cos x = (1 - \tan^2(x/2))/(1 + \tan^2(x/2)). \tag{6.5}$$

The default Maxima result for the first derivatives of $\sin x$, $\cos x$, and $\tan x$ is:

```
(%i1) map('diff,[sin(x),cos(x),tan(x)] )/del(x);
                                                2
(%o1)                        [cos(x), - sin(x), sec (x)]
```

Before using **gradef** to alter the return value of **diff** for these three functions, let's check that Maxima agrees with the trig "identities" displayed above. Variable amounts of expression simplification are needed to get what we want.

```
(%i2) trigsimp( 1 + tan(x)^2 );
                                          1
(%o2)                                  -------
                                          2
                                       cos (x)
(%i3) ( trigsimp( 2*tan(x/2)/(1 + tan(x/2)^2) ), trigreduce(%%) );
(%o3)                                  sin(x)
(%i4) ( trigsimp( (1-tan(x/2)^2)/(1 + tan(x/2)^2) ),
          trigreduce(%%), expand(%%) );
(%o4)                                  cos(x)
```

Recall that **trigsimp** will convert **tan**, **sec**, etc to **sin** and **cos** of the same argument. Recall also that **trigreduce** will convert an expression containing **cos(x/2)** and **sin(x/2)** into an expression containing **cos(x)** and **sin(x)**. Finally, remembering that $\sec x = 1/\cos x$, we see that Maxima has "passed the test".

Now let's use **gradef** to express the derivatives in terms of $\tan x$ and $\tan(x/2)$:

```
(%i5) gradef(tan(x), 1 + tan(x)^2 );
(%o5)                           tan(x)
(%i6) gradef(sin(x),(1-tan(x/2)^2 )/(1 + tan(x/2)^2 ) );
(%o6)                           sin(x)
(%i7) gradef(cos(x), -2*tan(x/2)/(1 + tan(x/2)^2 ) );
(%o7)                           cos(x)
```

Here we check the behavior:

```
(%i8) map('diff,[sin(x),cos(x),tan(x)] )/del(x);
                              2 x           x
                       1 - tan (-)     2 tan(-)
                                2           2         2
(%o8)                 [-----------, - -----------, tan (x) + 1]
                           2 x          2 x
                       tan (-) + 1   tan (-) + 1
                            2            2
```

6.2 Critical and Inflection Points of a Curve Defined by an Explicit Function

The **critical points** of a function $f(x)$ are the points x_j such that $f'(x_j) = 0$. We will use f' to indicate the first derivative, and f'' to indicate the second derivative. The **extrema** (ie., maxima and minima) are the values of the function at the critical points, provided the "slope" f' actually has a different sign on the opposite sides of the critical point.

Maximum: $f'(a) = 0$,	$f'(x)$	changes from $+$ to $-$,	$f''(a) < 0$
Minimum: $f'(a) = 0$,	$f'(x)$	changes from $-$ to $+$,	$f''(a) > 0$

If $f(x)$ is a function with a continuous second derivative, and if, as x increases through the value a, $f''(x)$ changes sign, then the plot of $f(x)$ has an **inflection point** at $x = a$ and $f''(a) = 0$. The **inflection point** requirement that $f''(x)$ changes sign at $x = a$ is equivalent to $f'''(a) \neq 0$. We consider some simple examples taken from Sect. 126 of Analytic Geometry and Calculus, by Lloyd L. Smail, Appleton-Century-Crofts, N.Y., 1953 (a reference which "dates" the writer of these notes!).

6.2.1 Example 1: A Polynomial

To find the maxima and minima of the function $f(x) = 2\,x^3 - 3\,x^2 - 12\,x + 13$, we use an "expression" (called **g**) rather than a Maxima function. We use **gp** (g prime) for the first derivative, and **gpp** (g double prime) as notation for the second derivative. Since we will want to make some simple plots, we use the package **qdraw**, available on the author's webpage, and discussed in detail in chapter five of these notes.

```
(%i1) load(draw)$
(%i2) load(qdraw);
                qdraw(...), qdensity(...), syntax: type qdraw();
(%o2)                       c:/work2/qdraw.mac
(%i3) g : 2*x^3 - 3*x^2 - 12*x + 13$
(%i4) gf : factor(g);
                                        2
(%o4)                     (x - 1) (2 x  - x - 13)
(%i5) gp : diff(g,x);
                                   2
(%o5)                         6 x  - 6 x - 12
(%i6) gpf : factor(gp);
(%o6)                        6 (x - 2) (x + 1)
(%i7) gpp : diff(gp,x);
(%o7)                            12 x - 6
(%i8) qdraw( ex(g,x,-4,4), key(bottom)  )$
```

Since the coefficients of the given polynomial are integral, we have tried out **factor** on both the given expression and its first derivative. We see from output **%o6** that the first derivative vanishes when **x = -1, 2** . We can confirm these critical points of the given function by using **qdraw** with the quick plotting argument **ex**, which gives us the plot:

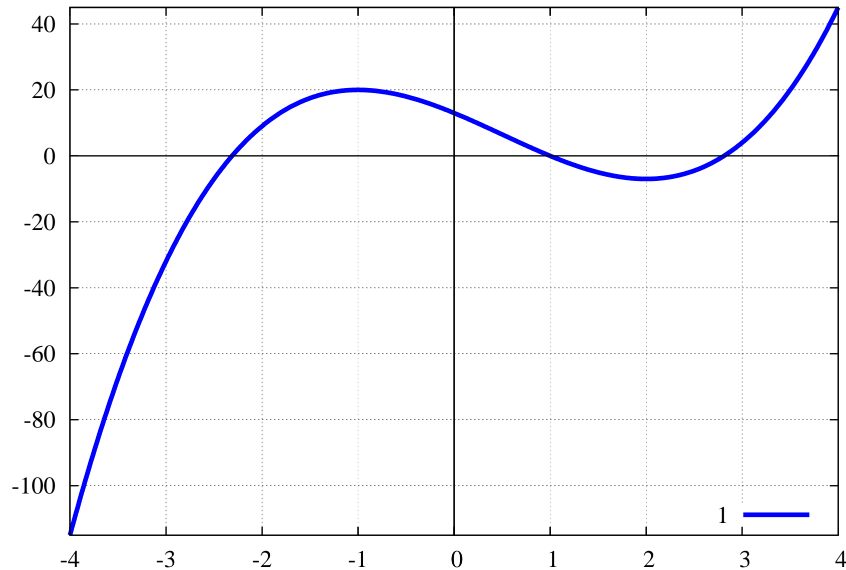

Figure 1: Plot of g over the range $[-4, 4]$

We can use the cursor on the plot to find that **g** takes on the value of roughly **-7** when **x = 2** and the value of about **20** when **x = -1**. We can use **solve** to find the roots of the equation **gp = 0**, and then evaluate the given expression **g**, as well as the second derivative **gpp** at the critical points found:

```
(%i9) solve(gp=0);
(%o9)                         [x = 2, x = - 1]
(%i10) [g,gpp],x=-1;
(%o10)                            [20, - 18]
(%i11) [g,gpp],x=2;
(%o11)                             [- 7, 18]
```

We next look for the **inflection points**, the points where the curvature changes sign, which is equivalent to the points where the second derivative vanishes.

```
(%i12) solve(gpp=0);
                                          1
(%o12)                               [x = -]
                                          2
(%i13) g,x=0.5;
(%o13)                                 6.5
```

We have found one inflection point, that is a point on a smooth curve where the curve changes from concave downward to concave upward, or visa versa (in this case the former).
We next plot **g, gp, gpp** together.

```
(%i14) qdraw2( ex([g,gp,gpp],x,-3,3), key(bottom),
                 pts([[-1,20]],ps(2),pc(magenta),pk("MAX") ),
              pts([[2,-7]],ps(2),pc(green),pk("MIN") ),
             pts([[0.5, 6.5]],ps(2),pk("INFLECTION POINT") ) )$
```

with the resulting plot:

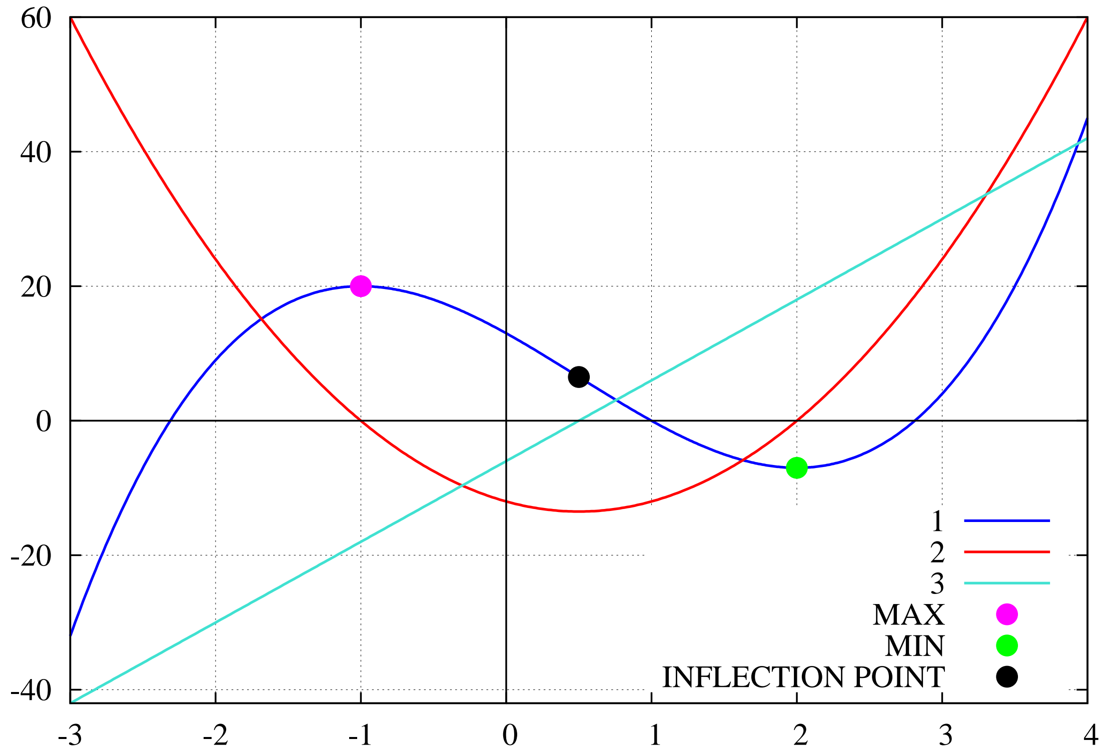

Figure 2: Plot of 1: g, 2: gp, and 3: gpp

The curve **g** is concave downward until **x = 0.5** and then is concave upward. Note that each successive differentiation tends to flatten out the kinks in the previous expression (function). In other words, **gp** is "flatter" than **g**, and **gpp** is "flatter" than **gp**. This behavior under **diff** is simply because each successive differentiation reduces by one the degree of the polynomial.

6.2.2 Automating Derivative Plots with plotderiv

In a separate text file **vcalc.mac** (available on the author's webpage) is a small homemade Maxima function called **plotderiv** which will plot a given expression together with as many derivatives as you want, using **qdraw** from Ch.5.

```
/* vcalc.mac */
plotderiv_syntax() :=
  disp("plotderiv(expr,x,x1,x2,y1,y2,numderiv)  constructs a list of
        the submitted expression expr and its first numderiv derivatives
        with respect to the independent variable, and then passes
        this list to qdraw(..). You need to have used load(draw)
        and load(qdraw) before using this function ")$

/*  version 1 commented out
plotderiv(expr,x,x1,x2,y1,y2,numderiv) :=
   block([plist],
     plist : [],
     for i thru numderiv do
        plist : cons(diff(expr,x,i), plist),
     plist : reverse(plist),
     plist : cons(expr, plist),
     display(plist),
     apply( 'qdraw, [ ex( plist,x,x1,x2 ),yr(y1,y2), key(bottom) ]))$ */
```

```
/* version 2  slightly more efficient  */
plotderiv(expr,x,x1,x2,y1,y2,numderiv) :=
   block([plist,aa],
     plist : [],
     aa[0] : expr,
     for i thru numderiv do (
        aa[i] : diff(aa[i-1],x),
        plist : cons(aa[i], plist)
         ),
     plist : reverse(plist),
     plist : cons(expr, plist),
     display(plist),
     apply( 'qdraw, [ ex( plist,x,x1,x2 ),yr(y1,y2), key(bottom) ]))$
```

We have provided two versions of this function, the first version commented out. If the expression to be plotted is a very complicated function and you are concerned with computing time, you can somewhat improve the efficiency of **plotderiv** by introducing a "hashed array" called **aa[j]**, say, to be able to simply differentiate the previous derivative expression. That is the point of version 2 which is not commented out.

We have added the line **display(plist)** to print a list containing the expression as well as all the derivatives requested. We test **plotderiv** on the expression **u^3**. Both **draw.lisp** and **qdraw.mac** must be loaded for this to work.

```
(%i15) load(vcalc)$
                vcalc.mac:   for syntax, type: vcalc_syntax();
 CAUTION: global variables set and used in this package:
       hhh1, hhh2, hhh3, uuu1, uuu2, uuu3, nnnsys
(%i16) plotderiv(u^3,u,-3,3,-27,27,4)$
                                         3     2
                          plist = [u , 3 u , 6 u, 6, 0]
```

which produces the plot:

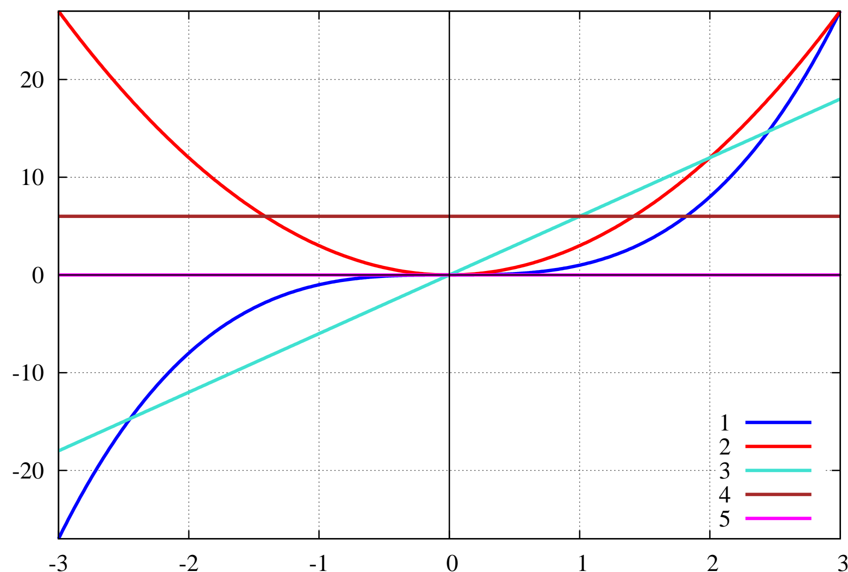

Figure 3: 1: $\mathbf{u^3}$, 2: $\mathbf{3\,u^2}$, 3: $\mathbf{6\,u}$, 4: $\mathbf{6}$, 5: $\mathbf{0}$

6.2.3 Example 2: Another Polynomial

We find the critical points, inflection points, and extrema of the expression $3\,x^4 - 4\,x^3$. We first look at the expression and its first two derivatives using **plotderiv**.

```
(%i17) plotderiv(3*x^4 - 4*x^3,x,-1,3,-5,5,2)$
                                4       3       3        2       2
              plist = [3 x  - 4 x , 12 x  - 12 x , 36 x  - 24 x]
```

which produces the plot:

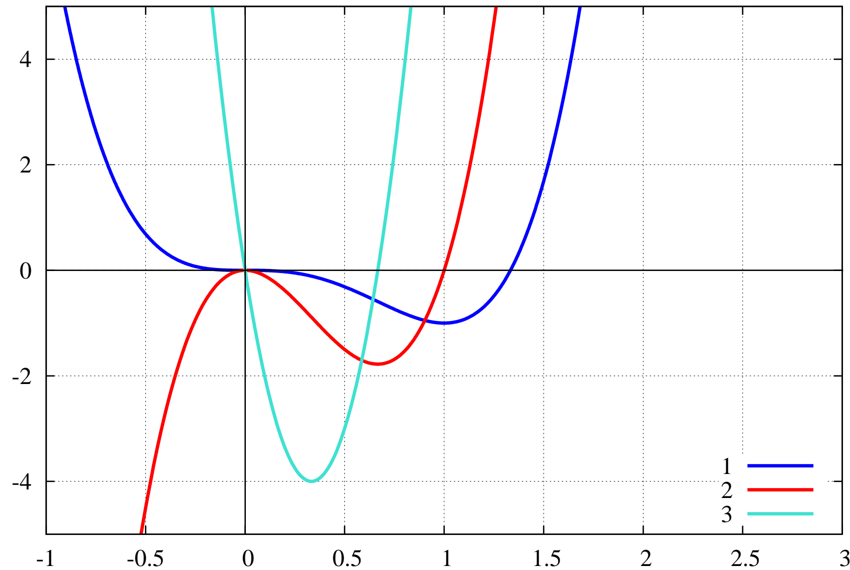

Figure 4: 1: $f = 3\,x^4 - 4\,x^3$, 2: f', 3: f''

We have inflection points at $x = 0,\ 2/3$ since the second derivative is zero at both points and changes sign as we follow the curve through those points. The curve changes from concave up to concave down passing through $x = 0$, and changes from concave down to concave up passing through $x = 2/3$. The following provides confirmation of our inferences from the plot:

```
(%i18) g:3*x^4 - 4*x^3$
(%i19) g1: diff(g,x)$
(%i20) g2 : diff(g,x,2)$
(%i21) g3 : diff(g,x,3)$
(%i22) x1 : solve(g1=0);
(%o22)                          [x = 0, x = 1]
(%i23) gcrit : makelist(subst(x1[i],g),i,1,2);
(%o23)                             [0, - 1]
(%i24) x2 : solve(g2=0);
                                    2
(%o24)                          [x = -, x = 0]
                                    3
(%i25) ginflec : makelist( subst(x2[i],g ),i,1,2 );
                                     16
(%o25)                             [- --, 0]
                                     27
(%i26) g3inflec : makelist( subst(x2[i],g3),i,1,2 );
(%o26)                            [24, - 24]
```

We have critical points where the first derivative vanishes at $x = 0,\ 1$. Since the first derivative does not change sign as the curve passes through the point $x = 0$, that point is neither a maximum nor a mimimum point. The point $x = 1$ is a minimum point since the first derivative changes sign from negative to positive as the curve passes through that point.

6.2.4 Example 3: $x^{2/3}$, Singular Derivative, Use of limit

Searching for points where a function has a minimum or maximum by looking at points where the first derivative is zero is useful as long as the first derivative is well behaved. Here is an example in which the given function of x has a minimum at $x = 0$, but the first derivative is singular at that point. Using the expression **g = x^(2/3)** with **plotderiv**:

```
(%i27) plotderiv(x^(2/3),x,-2,2,-2,2,1)$
```

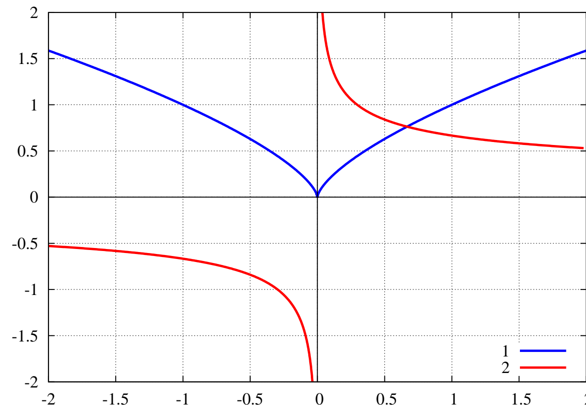

Figure 5: Plot of $1: \ f(x) = x^{2/3}, \ 2: \ f'$

We see that the first derivative is singular at $x = 0$ and approaches either positive or negative infinity, depending on your direction of approaching the point $x = 0$. We can see from the plot of $f(x) = x^{2/3}$ that the tangent line (the "slope") of the curve is negative for $x < 0$ and becomes increasingly negative (approaching minus infinity) as we approach the point $x = 0$ from the negative side. We also see from the plot of $f(x)$ that the tangent line ("slope") is positive for positive values of x and as we pass $x = 0$, moving from smaller x to larger x that the sign of the first derivative $f'(x)$ changes from $-$ to $+$, which is a signal of a local minimum of a function.

We can practice using the Maxima function **limit** with this example:

```
(%i28) gp : diff(x^(2/3),x);
                                       2
(%o28)                               ------
                                        1/3
                                     3 x
(%i29) limit(gp,x,0,plus);
(%o29)                                 inf
(%i30) limit(gp,x,0,minus);
(%o30)                               minf
(%i31) limit(gp,x,0);
(%o31)                                 und
```

The most frequent use of the Maxima **limit** function has the syntax

> Function: **limit (expr, x, val, dir)**
> Function: **limit (expr, x, val)**
> The first form computes the limit of **expr** as the real variable **x** approaches the value **val** from the direction **dir**. **dir** may have the value **plus** for a limit from above, **minus** for a limit from below.
> In the second form, **dir** is omitted, implying a "two-sided limit" is to be computed.
> **limit** uses the following special symbols: **inf** (positive infinity) and **minf** (negative infinity). On output it may also use **und** (undefined), **ind** (indefinite but bounded) and **infinity** (complex infinity).

Returning to our example, if we ignore the point $x = 0$, the slope is always decreasing, so the second derivative is always negative.

6.3 Tangent and Normal of a Point of a Curve Defined by an Explicit Function

The equation of a line with the form $y = m(x - x_0) + y_0$, where m, x_0, and y_0 are constants, is identically satisfied if we consider the point $(x, y) = (x_0, y_0)$. Hence this line passes through the point (x_0, y_0). The "slope" of this line (the first derivative) is the constant m.

Now we also assume that the point (x_0, y_0) is a point of a curve given by some explicit function of x, say $y = f(x)$; hence $y_0 = f(x_0)$. The first derivative of this function, evaluated at the point x_0 is the local "slope", which defines the **local tangent line**, $y = m(x - x_0) + y_0$, if we use for m the value of $f'(x_0)$, where the notation means first take the derivative for arbitrary x and then evaluate the derivative at $x = x_0$.

Let the two dimensional vector **tvec** be a vector parallel to the tangent line (at the point of interest) with components **(tx,ty)**, such that the vector makes an angle θ with the positive x axis. Then $\tan(\theta) = t_y/t_x = m = dy/dx$ is the slope of the tangent (line) at this point. Let the two dimensional vector **nvec** with components **(nx,ny)** be parallel to the line **perpendicular** to the tangent at the given point. This **normal** line will be perpendicular to the tangent line if the vectors **nvec** and **tvec** are perpendicular (i.e., "orthogonal").

In Maxima, we can represent vectors by lists:

```
(%i1) tvec : [tx,ty];
(%o1)                          [tx, ty]
(%i2) nvec : [nx, ny];
(%o2)                          [nx, ny]
```

Two vectors are "orthogonal" if the "dot product" is zero. A simple example will illustrate this general property. Consider the pair of unit vectors: **ivec = [1,0]**, parallel to the positive x axis and **jvec = [0,1]**, parallel to the positive y axis.

```
(%i3) ivec : [1,0]$
(%i4) jvec : [0,1]$
(%i5) ivec . jvec;
(%o5)                              0
```

Since Maxima allows us to use a **period** to find the dot product (inner product) of two lists (two vectors), we will ensure that the normal vector is "at right angles" to the tangent vector by requiring **nvec . tvec = 0**

```
(%i6) eqn : nvec . tvec = 0;
(%o6)                       ny ty + nx tx = 0
(%i7) eqn : ( eqn/(nx*ty), expand(%%) );
                                tx   ny
(%o7)                           -- + -- = 0
                                ty   nx
```

We represent the normal (the line perpendicular to the tangent line) at the point (x_0, y_0) as $y = m_n(x - x_0) + y_0$, where m_n is the slope of the normal line: **mn = ny/nx = - (1/(ty/tx)) = -1/m** .

In words, **the slope of the normal line is the negative reciprocal of the slope of the tangent line**.

Thus the equation of the **local normal** to the curve at the point (x_0, y_0) can be written as $y = -(x - x_0)/m + y_0$, where m is the slope of the local tangent.

6.3.1 Example 1: x^2

As an easy first example, let's use the function $f(x) = x^2$ and construct the tangent line and normal line to this plane curve at the point `x = 1, y = f(1) = 1`. (We have discussed this same example in Ch. 5). The first derivative is $2\,x$, and its value at $x = 1$ is 2. Thus the equation of the local tangent to the curve at `(x,y) = (1,1)` is $y = 2\,x - 1$. The equation of the local normal at the same point is $y = -x/2 + 3/2$.

We will plot the curve $y = x^2$, the local tangent, and the local normal together. Special care is needed to get the horizontal "canvas" width about `1.4` times the vertical height to achieve a geometrically correct plot; otherwise the "normal" line would not cross the tangent line at right angles.

```
(%i8) qdraw( ex([x^2,2*x-1,-x/2+3/2],x,-1.4,2.8), yr(-1,2) ,
             pts( [ [1,1]],ps(2) ) )$
```

The plot looks like:

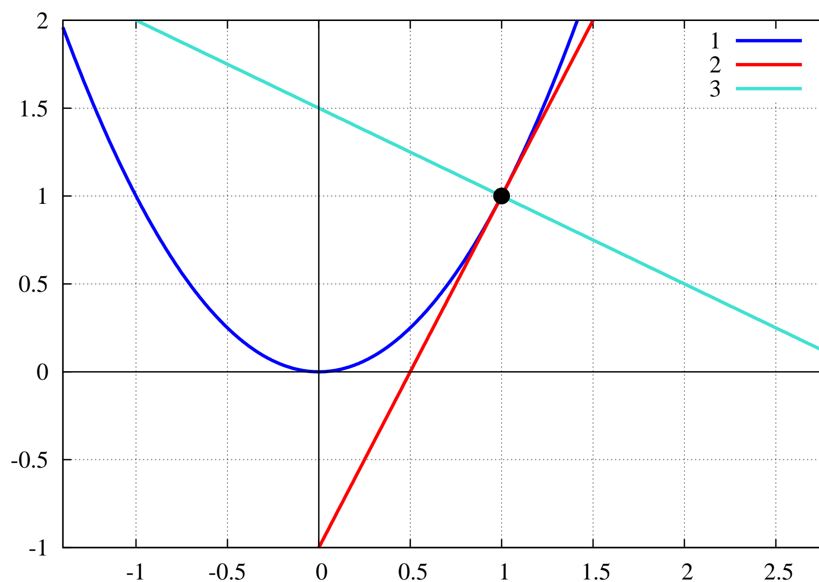

Figure 6: Tangent and Normal to x^2 at point $(1,1)$

We can see directly from the plot that the slope of the tangent line has the value 2. Remember that we can think of "slope" as being the number derived from dividing the "rise" by the "run". Thus if we choose a "run" to be the x interval `(0,1)`, from the plot this corresponds to the "rise" of $1 - (-1) = 2$. Hence "rise/run" is 2.

6.3.2 Example 2: $\ln(x)$

Our next example repeats the analysis of Example 1 for the function $\ln(x)$, using the point $(x = 2, y = \ln(2))$. Maxima's natural log function is written `log`. Maxima does not have a "log to the base 10 function", although you can "roll your own" with a definition like `log10(x) := log(x)/log(10)`, which is equivalent to `log(x)/2.303`. Thus in Maxima, `log(10) = 2.303, log(2) = 0.693`, etc. We are constructing the local tangent and normal at the point $(x = 2, y = 0.693)$. The derivative of the natural log is

```
(%i9) diff(log(x),x);
                                    1
(%o9)                               -
                                    x
```

so the "slope" of our curve (and the tangent) at the chosen point is `1/2`, and the slope of the local normal is `-2`. The equation of the tangent is then $(y - \ln(2)) = (1/2)\,(x - 2)$, or $y = x/2 - 0.307$. The equation of the normal

is $(\mathbf{y} - \ln(\mathbf{2})) = -\mathbf{2}\,(\mathbf{x} - \mathbf{2})$, or $\mathbf{y} = -\mathbf{2}\,\mathbf{x} + \mathbf{4.693}$. In order to get a decent plot, we need to stay away from the singular point $\mathbf{x} = \mathbf{0}$, so we start the plot with $\mathbf{x}$ a small positive number. We choose to use the x-axis range $(\mathbf{0.1}, \mathbf{4})$, then $\boldsymbol{\Delta}\mathbf{x} = \mathbf{3.9}$. Since we want the "canvas width" $\boldsymbol{\Delta}\mathbf{x} \approx \mathbf{1.4}\,\boldsymbol{\Delta}\mathbf{y}$, we need $\boldsymbol{\Delta}\mathbf{y} = \mathbf{2.786}$, which is satisfied if we choose the y-axis range to be $(-\mathbf{1}, \mathbf{1.786})$.

```
(%i10) qdraw(key(bottom), yr(-1,1.786),
         ex([log(x), x/2 - 0.307, -2*x + 4.693],x,0.1,4),
         pts( [ [2,0.693]],ps(2) )  );
```

which produces the plot:

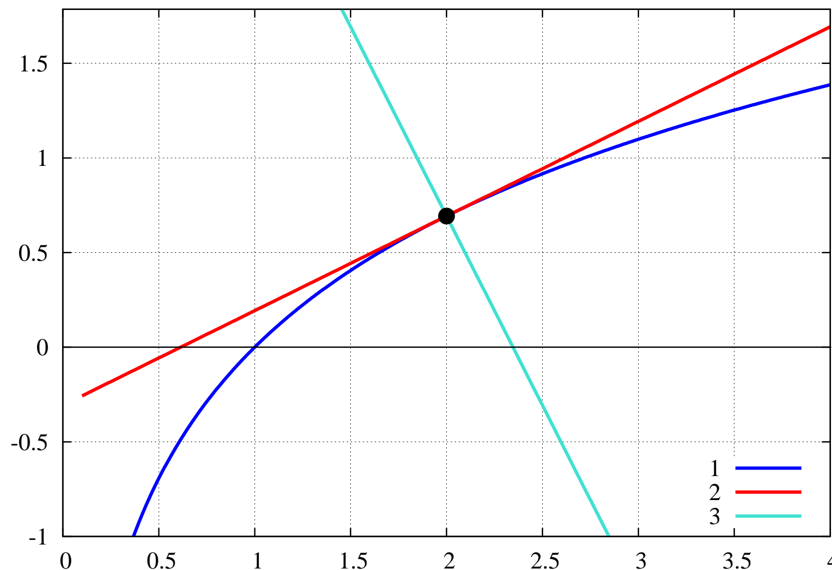

Figure 7: Tangent and Normal to $\ln(\mathbf{x})$ at point $(\mathbf{2}, \ln(\mathbf{2}))$

6.4 Maxima and Minima of a Function of Two Variables

You can find proofs of the following criteria in calculus texts.

If $\mathbf{f}(\mathbf{x}, \mathbf{y})$ and its first derivatives are continuous in a region including the point $(\mathbf{a}, \mathbf{b})$, a necessary condition that $\mathbf{f}(\mathbf{a}, \mathbf{b})$ shall be an extreme (maximum or minimum) value of the function $\mathbf{f}(\mathbf{x}, \mathbf{y})$ is that (I):

$$\frac{\partial f(a,b)}{\partial x} = 0, \quad \frac{\partial f(a,b)}{\partial y} = 0 \tag{6.6}$$

in which the notation means first take the partial derivatives for arbitrary $\mathbf{x}$ and $\mathbf{y}$ and then evaluate the resulting derivatives at the point $(\mathbf{x}, \mathbf{y}) = (\mathbf{a}, \mathbf{b})$.

Examples show that these two conditions (I) do not guarantee that the function actually takes on an extreme value at the point $(\mathbf{a}, \mathbf{b})$, although in many practical problems the existence and nature of an extreme value is often evident from the problem itself and no extra test is needed; all that is needed is the **location** of the extreme value.

If condition (I) is satisfied and if, in addition, at the point $(\mathbf{a}, \mathbf{b})$ we have (II):

$$\Delta = \left(\frac{\partial^2 f}{\partial x^2}\right)\left(\frac{\partial^2 f}{\partial y^2}\right) - \left(\frac{\partial^2 f}{\partial x\,\partial y}\right)^2 > 0 \tag{6.7}$$

then $\mathbf{f}(\mathbf{x}, \mathbf{y})$ will have a maximum value or a minimum value given by $\mathbf{f}(\mathbf{a}, \mathbf{b})$ according as $\partial^2 f/\partial x^2$ (or $\partial^2 f/\partial y^2$) is negative or positive for $\mathbf{x} = \mathbf{a}, \mathbf{y} = \mathbf{b}$. If condition (I) holds and $\boldsymbol{\Delta} < \mathbf{0}$ then $\mathbf{f}(\mathbf{a}, \mathbf{b})$ is neither a maximum nor a minimum; if $\boldsymbol{\Delta} = \mathbf{0}$ the test fails to give any information.

6.4.1 Example 1: Minimize the Area of a Rectangular Box of Fixed Volume

Let the sides of a "rectangular box" (rectangular prism, or cuboid, if you wish) of fixed volume v be x, y, and z. We wish to determine the shape of this box (for fixed v) which minimizes the surface area $s = 2\,(x\,y + y\,z + z\,x)$. We can use the fixed volume relation to express z as a function of x and y and then achieve an expression for the surface area which only depends of the two variables x and y. We then know that a necessary condition that the surface area be an extremum is that the two equations of condition I above be satisfied. We need to expose the hidden dependence of z on x and y via the volume constraint before we can correctly derive the two equations of condition I.

You probably already know the answer to this shape problem is a cube, in which all sides are equal in length, and the length of a side is the same as the cube root of the volume.

We require three equations to be satisfied, the first being the fixed volume $v = x\,y\,z$, and the other two being the equations of condition I above. There are multiple paths to such a solution in Maxima. Perhaps the simplest is to use **solve** to generate a solution of the three equations for the variables (x, y, z), although the solutions returned will include non-physical solutions and we must select the physically realizable solution.

```
(%i1) eq1 : v = x*y*z;
(%o1)                              v = x y z
(%i2) solz : solve(eq1,z);
                                         v
(%o2)                              [z = ---]
                                        x y
(%i3) s : ( subst(solz,2*(x*y  + y*z + z*x) ), expand(%%) );
                                        2 v   2 v
(%o3)                          2 x y + --- + ---
                                         y     x
(%i4) eq2 : diff(s,x)=0;
                                        2 v
(%o4)                            2 y - --- = 0
                                         2
                                        x
(%i5) eq3 : diff(s,y) = 0;
                                        2 v
(%o5)                            2 x - --- = 0
                                         2
                                        y
(%i6) solxyz: solve([eq1,eq2,eq3],[x,y,z]);
                  1/3       1/3       1/3
(%o6) [[x = v   , y = v   , z = v   ],
                 1/3                             1/3
          2 v                 (sqrt(3) %i + 1) v
[x = --------------, y = - ---------------------,
     sqrt(3) %i - 1                  2
                            1/3                 1/3
       (sqrt(3) %i + 1) v                  2 v
z = - ---------------------], [x = - --------------,
                2                     sqrt(3) %i + 1
                          1/3                        1/3
    (sqrt(3) %i - 1) v          (sqrt(3) %i - 1) v
y = ---------------------, z = ---------------------]]
              2                          2
```

Since the physical solutions must be real, the first sublist is the physical solution which implies the cubical shape as the answer.

An alternative solution path is to solve **eq2** above for y as a function of x and v and use this to eliminate y from **eqn3**.

```
(%i7) soly : solve(eq2,y);
                                         v
(%o7)                               [y = --]
                                          2
                                         x
(%i8) eq3x : ( subst(soly, eq3),factor(%%) );
                                       3
                                 2 x (x  - v)
(%o8)                          - ------------ = 0
                                      v
```

The factored form **%o7** shows that, since $x \neq 0$, the solution must have $x = v^{1/3}$. We can then use this solution for x to generate the solution for y and then for z.

```
(%i9) solx : x = v^(1/3);
                                         1/3
(%o9)                               x = v
(%i10) soly : subst(solx, soly);
                                         1/3
(%o10)                              [y = v   ]
(%i11) subst( [solx,soly[1] ], solz );
                                         1/3
(%o11)                             [z = v   ]
```

We find that $\Delta > 0$ and that the second derivative of $s(x, y)$ with respect to x is positive, as needed for the cubical solution to define a minimum surface solution.

```
(%i12) delta : ( diff(s,x,2)*diff(s,y,2) - diff(s,x,1,y,1), expand(%%) );
                                        2
                                    16 v
(%o12)                              ----- - 2
                                     3  3
                                    x  y
(%i13) delta : subst([x^3=v,y^3=v], delta );
(%o13)                                 14
(%i14) dsdx2 : diff(s,x,2);
                                       4 v
(%o14)                                 ---
                                        3
                                       x
(%i15) dsdy2 : diff(s,y,2);
                                       4 v
(%o15)                                 ---
                                        3
                                       y
```

6.4.2 Example 2: Maximize the Cross Sectional Area of a Trough

A long rectangular sheet of aluminum of width $\mathbf{L}$ is to be formed into a flat bottom trough by bending both a left and right hand length $\mathbf{x}$ up by an angle $\boldsymbol{\theta}$ with the horizontal. A cross section view is then:

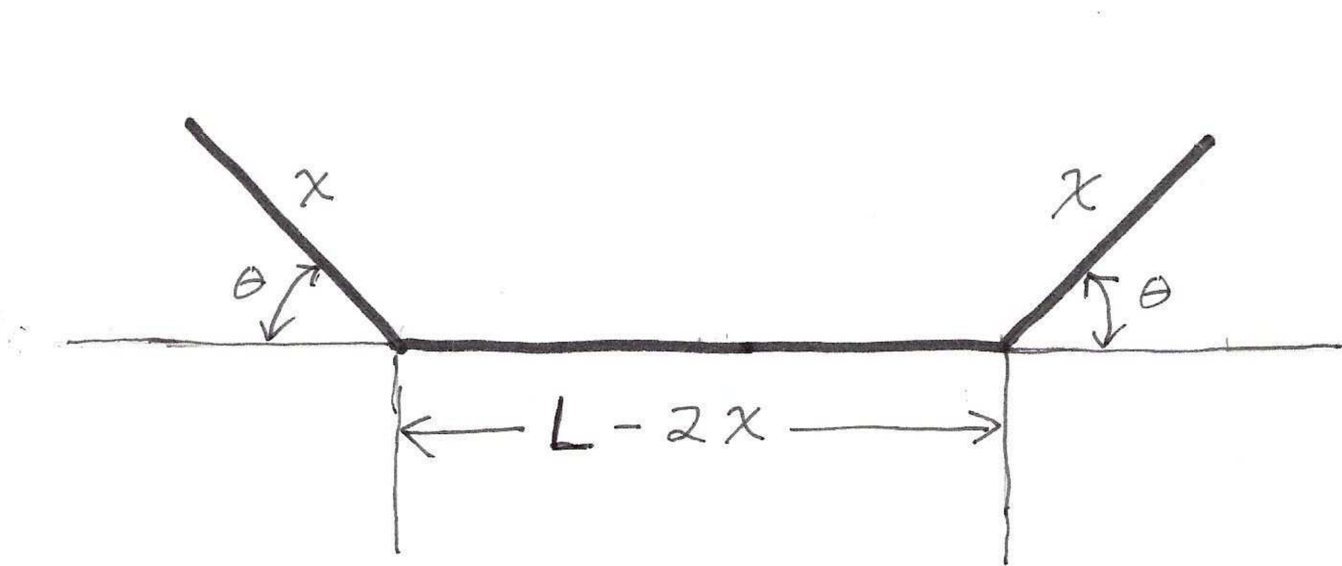

Figure 8: Flat Bottomed Trough

The lower base is $\mathbf{l} = \mathbf{L} - \mathbf{2\,x}$, the upper base is $\mathbf{u} = \mathbf{l} + \mathbf{2\,x\cos(\theta)}$. The altitude is $\mathbf{h} = \mathbf{x\sin(\theta)}$. The area of a trapezoid whose parallel sides are $\mathbf{l}$ and $\mathbf{u}$ and whose altitude is $\mathbf{h}$ is $(\mathbf{h}/\mathbf{2})\,(\mathbf{l} + \mathbf{u})$ (ie., the height times the average width: Exercise: check this by calculating the area of a rectange $\mathbf{h\,u}$, where $\mathbf{u}$ is the larger base, and then subtracting the area of the two similar right triangles on the left and right hand sides.) Hence the area of the cross section (called **s** in our Maxima work) is $A = L\,x\,\sin(\theta) - 2\,x^2\,\sin(\theta) + x^2\,\sin(\theta)\,\cos(\theta)$

Using condition I equations, we proceed to find the extremum solutions for $\mathbf{x}$ and $\boldsymbol{\theta}$ (which is called **th** in our Maxima work). Our method is merely one possible path to a solution.

```
(%i1) s : L*x*sin(th) - 2*x^2*sin(th) + x^2*sin(th)*cos(th);
                                                2             2
(%o1)           sin(th) x L + cos(th) sin(th) x  - 2 sin(th) x
(%i2) dsdx : ( diff(s,x), factor(%%) );
(%o2)                    sin(th) (L + 2 cos(th) x - 4 x)
```

We see that **one way** we can get **dsdx** to be zero is to set $\sin(\theta) = 0$, however this would mean the angle of bend was zero degrees, which is not the solution of interest. Hence our first equation comes from setting the second factor of **dsdx** equal to zero.

```
(%i3) eq1 : dsdx/sin(th) = 0;
(%o3)                         L + 2 cos(th) x - 4 x = 0
(%i4) solx : solve(eq1,x);
                                            L
(%o4)                            [x = - -------------]
                                        2 cos(th) - 4
(%i5) dsdth : ( diff(s,th), factor(%%) );
                                     2          2
(%o5)         x (cos(th) L - sin (th) x + cos (th) x - 2 cos(th) x)
```

We see that we can get **dsdth** to be zero if we set $x = 0$, but this is not the physical situation we are examining. We assume (of course) that $x \neq 0$, and arrive at our second equation of condition I by setting the second factor of **dsdth** equal to zero.

```
(%i6) eq2 : dsdth/x = 0;
                               2            2
(%o6)        cos(th) L - sin (th) x + cos (th) x - 2 cos(th) x = 0
(%i7) eq2 : ( subst(solx,eq2), ratsimp(%%) );
                           2          2
                     (sin (th) + cos (th) - 2 cos(th)) L
(%o7)                ----------------------------------- = 0
                               2 cos(th) - 4
```

The only way we can satisfy **eq2** is to set the numerator of the left hand side equal to zero, and divide out the factor **L** which is positive:

```
(%i8) eq2 : num(lhs(eq2) )/L = 0;
                            2          2
(%o8)                  sin (th) + cos (th) - 2 cos(th) = 0
(%i9) eq2 : trigsimp(eq2);
(%o9)                           1 - 2 cos(th) = 0
(%i10) solcos : solve( eq2, cos(th) );
                                               1
(%o10)                            [cos(th) = -]
                                               2
(%i11) solx : subst(solcos, solx);
                                            L
(%o11)                                 [x = -]
                                            3
(%i12) solth : solve(solcos,th);
`solve' is using arc-trig functions to get a solution.
Some solutions will be lost.
                                           %pi
(%o12)                               [th = ---]
                                            3
```

6.5 Tangent and Normal of a Point of a Curve Defined by an Implicit Function

In the following, Maxima assumes that **y** is independent of **x**:

```
(%i1) diff(x^2 + y^2,x);
(%o1)                                  2 x
```

We can indicate explicitly that **y** depends on **x** for purposes of taking this derivative by replacing **y** by **y(x)**.

```
(%i2) diff(x^2 + y(x)^2,x);
                                        d
(%o2)                          2 y(x) (-- (y(x))) + 2 x
                                        dx
```

Instead of using the functional notation to indicate dependency, we can use the **depends** function before taking the derivative.

```
(%i3) depends(y,x);
(%o3)                              [y(x)]
(%i4) g : diff(x^2 + y^2, x);
                                     dy
(%o4)                            2 y -- + 2 x
                                     dx
(%i5) grind(g)$
2*y*'diff(y,x,1)+2*x$
(%i6) gs1 : subst('diff(y,x) = x^3, g );
                                   3
(%o6)                            2 x  y + 2 x
```

In **%i4** we defined **g** as the derivative (with respect to **x**) of the expression **x^2 + y^2**, after telling Maxima that the symbol **y** is to be considered dependent on the value of **x**. Since Maxima, as yet, has no specific information about the nature of the dependence of **y** on **x**, the output is expressed in terms of the "noun form" of **diff**, which Maxima's pretty print shows as $\frac{dy}{dx}$. To see the "internal" form, we again use the **grind** function, which shows the explicit noun form **'diff(y,x,1)**. This is useful to know if we want to later replace that unknown derivative with a known result, as we do in input **%i6**.

However, Maxima doesn't do anything creative with the derivative substitution done in **%i6**, like working backwards to what the explicit function of **x** the symbol **y** might stand for (different answers differ by a constant). The output **%o6** still contains the symbol **y**.

Two ways to later implement our knowledge of **y(x)** are shown in steps **%i7** and **%i8**, which use the **ev** function. (In Chapter 1 we discussed using **ev** for making substitutions, although the use of **subst** is more generally recommended for that job.)

```
(%i7) ev(g,y=x^4/4,diff);
                                  7
                                 x
(%o7)                           -- + 2 x
                                 2
(%i8) ev(g,y=x^4/4,nouns);
                                  7
                                 x
(%o8)                           -- + 2 x
                                 2
(%i9) y;
(%o9)                                y
(%i10) g;
                                     dy
(%o10)                           2 y -- + 2 x
                                     dx
```

We see that Maxima does not bind the symbol **y** to anything when we call **ev** with an equation like **y = x^4/4**, and that the binding of **g** has not changed.

A list which reminds you of all dependencies in force is **dependencies**. The Maxima function **diff** is the only core function which makes use of the **dependencies** list. The functions **integrate** and **laplace** do not use the **depends** assignments; one must indicate the dependence explicitly by using functional notation.

In the following, we first ask for the contents of the **dependencies** list and then ask Maxima to remove the above dependency of **y** on **x**, using **remove**, then check the list contents again, and carry out the previous differentiation with Maxima no longer assuming that **y** depends on **x**.

```
(%i11) dependencies;
(%o11)                              [y(x)]
(%i12) remove(y, dependency);
(%o12)                               done
(%i13) dependencies;
(%o13)                                []
(%i14) diff(x^2 + y^2,x);
(%o14)                               2 x
```

On can also remove the properties associated with the symbol **y** by using **kill (y)**, although this is more drastic than using **remove**.

```
(%i15) depends(y,x);
(%o15)                              [y(x)]
(%i16) dependencies;
(%o16)                              [y(x)]
(%i17) kill(y);
(%o17)                               done
(%i18) dependencies;
(%o18)                                []
(%i19) diff(x^2+y^2,x);
(%o19)                               2 x
```

There are many varieties of using the **kill** function. The way we are using it here corresponds to the syntax:

> Function: **kill (a_1, ..., a_n)**
> Removes all bindings (value, function, array, or rule) from the arguments **a_1, ..., a_n**. An argument **a_k** may be a symbol or a single array element.

The list **dependencies** is one of the lists Maxima uses to hold information introduced during a work session. You can use the command **infolists** to obtain a list of the names of all of the information lists in Maxima.

```
(%i1) infolists;
(%o2) [labels, values, functions, macros, arrays, myoptions, props, aliases,
                    rules, gradefs, dependencies, let_rule_packages, structures]
```

When you first start up Maxima, each of the above named lists is empty.

```
(%i2) functions;
(%o2)                                 []
```

6.5.1 Tangent of a Point of a Curve Defined by $f(x, y) = 0$

Suppose some plane curve is defined by the equation $f(x, y) = 0$. Every point (x, y) belonging to the curve must satisfy that equation. For such pairs of numbers, changing x forces a change in y so that the equation of the curve is still satisfied. When we start plotting tangent lines to plane curves, we will need to evaluate the change in y, given a change in x, such that the numbers (x, y) are always points of the curve. Let's then regard y as depending on x via the equation of the curve. Given that the equation $f(x, y) = 0$ is valid, we obtain another valid equation by taking the derivative of both sides of the equation with respect to x. Let's work just on the left hand side of the resulting equation for now:

```
(%i1) depends(y,x);
(%o1)                              [y(x)]
(%i2) diff(f(x,y),x);
                                d
(%o2)                           -- (f(x, y))
                                dx
```

We can make progress by assigning values to the partial derivative of $f(x, y)$ with respect to the first argument x and also to the partial derivative of $f(x, y)$ with respect to the second argument y, using the **gradef** function. The assigned values can be explicit mathematical expressions or symbols. We are going to adopt the symbol **dfdx** to stand for the partial derivative

$$dfdx = \left(\frac{\partial f(x, y)}{\partial x}\right)_y ,$$

where the y subscript means "treat y as a constant when evaluating this derivative".

Likewise, we will use the symbol **dfdy** to stand for the partial derivative of $f(x, y)$ with respect to y, holding x constant:

$$dfdy = \left(\frac{\partial f(x, y)}{\partial y}\right)_x .$$

```
(%i3) gradef(f(x,y), dfdx, dfdy );
(%o3)                              f(x, y)
(%i4) g : diff( f(x,y), x );
                                    dy
(%o4)                          dfdy -- + dfdx
                                    dx
(%i5) grind(g)$
dfdy*'diff(y,x,1)+dfdx$
(%i6) g1 : subst('diff(y,x) = dydx, g);
(%o6)                         dfdy dydx + dfdx
```

We have adopted the symbol **dydx** for the "rate of change" of y as x varies, subject to the constraint that the numbers (x, y) always satisfy the equation of the curve $f(x, y) = 0$, which implies that **g1 = 0**.

```
(%i7) soln s : solve(g1=0, dydx);
                                       dfdx
(%o7)                         [dydx = - ----]
                                       dfdy
```

We see that Maxima knows enough about the rules for differentiation to allow us to get to a famous calculus formula. If x and y are constrained by the equation $f(x, y) = 0$, then the "slope" of the local tangent to the point (x, y) is

$$\frac{d\,y}{d\,x} = -\frac{\left(\frac{\partial f(x,y)}{\partial x}\right)_y}{\left(\frac{\partial f(x,y)}{\partial y}\right)_x}. \tag{6.8}$$

Of course, this formal expression has no meaning at a point where the denominator is zero.

In your calculus book you will find the derivation of the differentiation rule embodied in our output **%o4** above:

$$\frac{d}{d\,x} f(x, y(x)) = \left(\frac{\partial f(x, y)}{\partial x}\right)_y + \left(\frac{\partial f(x, y)}{\partial y}\right)_x \frac{d\,y(x)}{d\,x} \tag{6.9}$$

Remember that Maxima knows only what the code writers put in; we normally assume that the correct laws of mathematics are encoded as an aid to calculation. If Maxima were not consistent with the differentiation rule Eq. (6.9), then a bug would be present and would have to be removed.

The result Eq.(6.8) provides a way to find the slope (which we call **m**) at a general point (x, y).

This "slope" should be evaluated at the curve point (x_0, y_0) of interest to get the tangent and normal in numerical form. Recall our discussion in the first part of Section(6.3) where we derived the relation between the slopes of the tangent and normal. If the numerical value of the slope is m_0, then the tangent (line) is the equation $y = m_0\,(x - x_0) + y_0$, and the normal (line) is the equation $y = -(x - x_0)/m_0 + y_0$

A Function which Solves for dy/dx given that $f(x,y) = 0$

Given an equation relating x and y, we assume we can rewrite that equation in the form $f(x,y) = 0$, and then define the following function:

```
(%i1) dydx(expr,x,y) := -diff(expr,x)/diff(expr,y);
                                          - diff(expr, x)
(%o1)                    dydx(expr, x, y) := ---------------
                                            diff(expr, y)
```

As a first example, consider finding dy/dx given that $x^3 + y^3 = 1$.

```
(%i2) dydx( x^3+y^3-1,x,y);
                                       2
                                      x
(%o2)                               - --
                                       2
                                      y
```

Next find dy/dx given that $\cos(x^2 - y^2) = y\,cos(x)$.

```
(%i3) r1 : dydx( cos(x^2 - y^2) - y*cos(x),x,y );
                                   2    2
                        - 2 x sin(y   - x ) - sin(x) y
(%o3)                   -----------------------------
                                    2    2
                         - 2 y sin(y   - x ) - cos(x)
(%i4) r2 : (-num(r1))/(-denom(r1));
                                  2    2
                         2 x sin(y   - x ) + sin(x) y
(%o4)                   ---------------------------
                                   2    2
                          2 y sin(y   - x ) + cos(x)
```

Maxima does not simplify `r1` by cancelling the minus signs automatically, and we have resorted to dividing the negative of the numerator by the negative of the denominator(!) to get the form of `r2`. Notice also that Maxima has chosen to write $\sin(y^2 - x^2)$ instead of $\sin(x^2 - y^2)$.

Finally, find dy/dx given that $3\,y^4 + 4\,x - x^2\sin(y) - 4 = 0$.

```
(%i5) dydx(3*y^4 +4*x -x^2*sin(y) - 4, x, y );
                              2 x sin(y) - 4
(%o5)                        -----------------
                                3    2
                            12 y  - x  cos(y)
```

6.5.2 Example 1: Tangent and Normal of a Point of a Circle

As an easy first example, lets consider a circle of radius r defined by the equation $x^2 + y^2 = r^2$. Let's choose $r = 1$. Then $f(x,y) = x^2 + y^2 - 1 = 0$ defines the circle of interest, and we use the "slope" function above to calculate the slope m for a general point (x,y).

```
(%i1) dydx(expr,x,y) := -diff(expr,x)/diff(expr,y)$
(%i2) f:x^2 + y^2-1$
(%i3) m : dydx(f,x,y);
                                       x
(%o3)                                - -
                                       y
```

We then choose a point of the circle, evaluate the slope at that point, and construct the tangent and normal at that point.

```
(%i4) fpprintprec:8$
(%i5) [x0,y0] : [0.5,0.866];
(%o5)                              [0.5, 0.866]
(%i6) m : subst([x=x0,y=y0],m );
(%o6)                               - 0.577367
(%i7) tangent : y = m*(x-x0) + y0;
(%o7)                     y = 0.866 - 0.577367 (x - 0.5)
(%i8) normal : y = -(x-x0)/m + y0;
(%o8)                       y = 0.866 - 1.732 (0.5 - x)
```

We can then use **qdraw** to show the circle with both the tangent and normal.

```
(%i9) load(draw);
(%o9)   C:/PROGRA~1/MAXIMA~4.0/share/maxima/5.15.0/share/draw/draw.lisp
(%i10) load(qdraw);
               qdraw(...), qdensity(...), syntax: type qdraw();

(%o10)                          c:/work2/qdraw.mac
(%i11) ratprint:false$
(%i12) qdraw(key(bottom),ipgrid(15),
          imp([ f = 0,tangent,normal],x,-2.8,2.8,y,-2,2 ),
           pts([ [0.5,0.866]],ps(2) ) )$
```

The result is

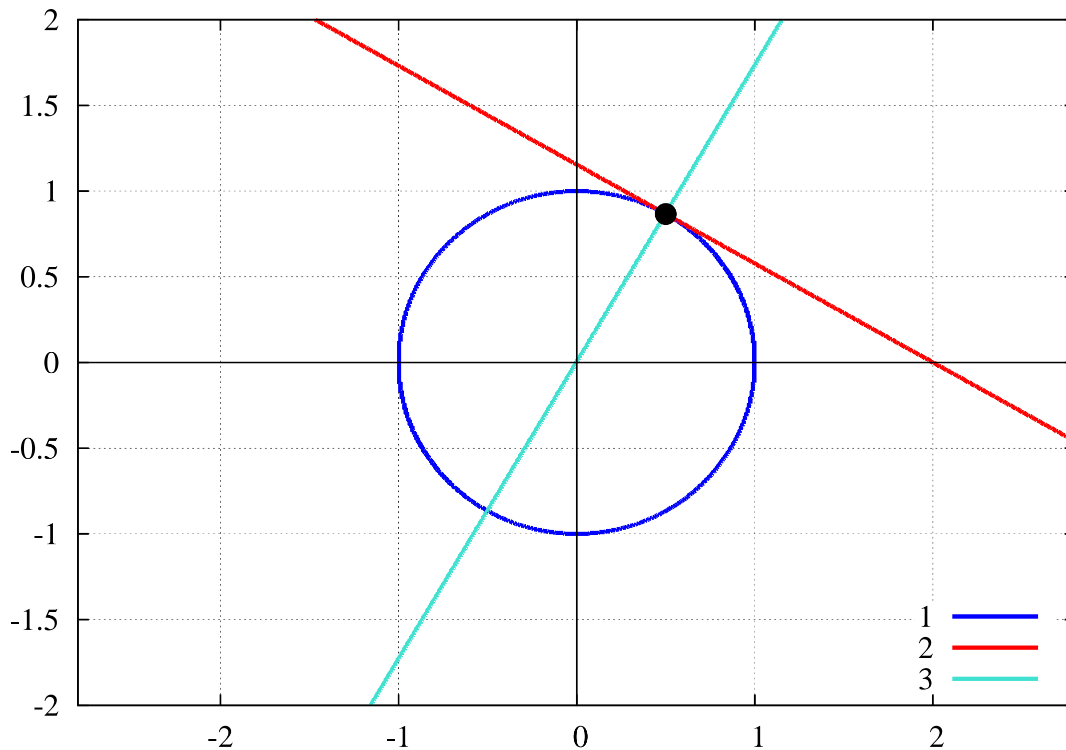

Figure 9: Tangent and Normal to $x^2 + y^2 = 1$ at point $(0.5, 0.866)$

6.5.3 Example 2: Tangent and Normal of a Point of the Curve $\sin(2\,x)\ \cos(y) = 0.5$

A curve is defined by $\mathbf{f(x, y) = \sin(2\,x)\cos(y) - 0.5 = 0}$, with $\mathbf{0 \le x \le 2}$ and $\mathbf{0 \le y \le 2}$. Using Eq. (6.8) we calculate the slope of the tangent to this curve at some point $\mathbf{(x, y)}$ and then specialize to the point $\mathbf{(x = 1, y = y_0)}$ with $\mathbf{y_0}$ to be found:

```
(%i13) f : sin(2*x)*cos(y) - 0.5$
(%i14) m : dydx(f,x,y);
                                 2 cos(2 x) cos(y)
(%o14)                           -----------------
                                  sin(2 x) sin(y)
(%i15) s1 : solve(subst(x=1,f),y);
`solve' is using arc-trig functions to get a solution.
Some solutions will be lost.
                                            1
(%o15)                        [y = acos(--------)]
                                         2 sin(2)
(%i16) fpprintprec:8$
(%i17) s1 : float(s1);
(%o17)                          [y = 0.988582]
(%i18) m : subst([ x=1, s1[1] ], m);
                                 1.3166767 cos(2)
(%o18)                           ----------------
                                      sin(2)
(%i19) m : float(m);
(%o19)                              - 0.602587
(%i20) mnorm : -1/m;
(%o20)                               1.6595113
(%i21) y0 : rhs( s1[1] );
(%o21)                               0.988582
(%i22) qdraw( imp([f = 0, y - y0 = m*(x - 1),
              y - y0 = mnorm*(x - 1) ],x,0,2,y,0,1.429),
             pts( [ [1,y0] ], ps(2) ), ipgrid(15))$
```

which produces the plot

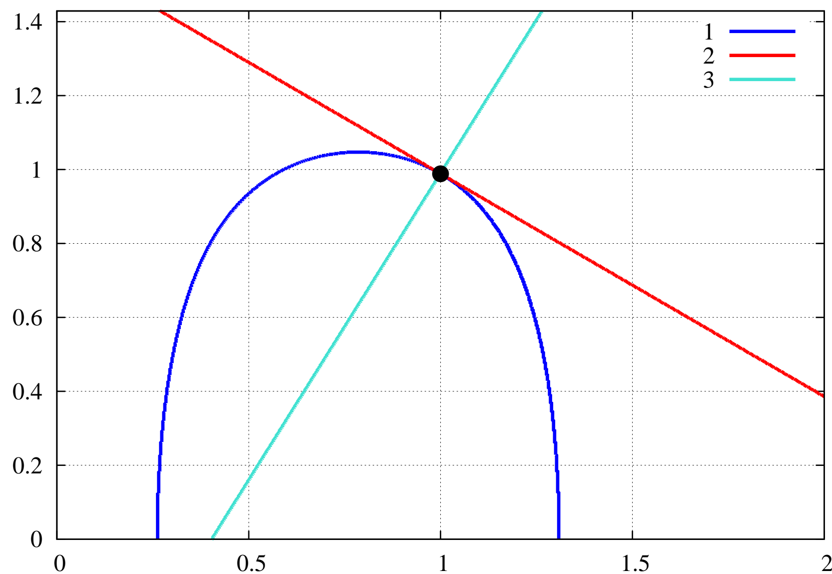

Figure 10: Tangent and Normal to $\mathbf{\sin(2\,x)\cos(y) = 0.5}$ at point $\mathbf{(1, 0.988)}$

6.5.4 Example 3: Tangent and Normal of a Point on a Parametric Curve: $x = \sin(t), y = \sin(2\,t)$

This example is the parametric curve example we plotted in Ch. 5. If we divide the differential of y by the differential of x, the common factor of $d\,t$ will cancel out and we will have an expression for the slope of the tangent to the curve at a point determined by the value the parameter t, which in this example must be an angle expressed in radians (as usual in calculus).

We then specialize to a point on the curve corresponding to $x = 0.8$, with $0 \le t \le \pi/2$, which we solve for:

```
(%i23) m : diff(sin(2*t))/diff(sin(t));
                                  2 cos(2 t)
(%o23)                            ----------
                                    cos(t)
(%i24) x0 : 0.8;
(%o24)                               0.8
(%i25) tsoln : solve(x0 = sin(t), t);
                                          4
(%o25)                            [t = asin(-)]
                                          5
(%i26) tsoln : float(tsoln);
(%o26)                           [t = 0.927295]
(%i27) t0 : rhs( tsoln[1] );
(%o27)                             0.927295
(%i28) m : subst( t = t0, m);
(%o28)                            - 0.933333
(%i29) mnorm : -1/m;
(%o29)                             1.0714286
(%i30) y0 : sin(2*t0);
(%o30)                               0.96
(%i31) qdraw( xr(0, 2.1), yr(0,1.5), ipgrid(15),nticks(200),
              para(sin(t),sin(2*t),t,0,%pi/2, lc(brown) ),
             ex([y0+m*(x-x0),y0+mnorm*(x-x0)],x,0,2.1 ),
            pts( [ [0.8, 0.96]],ps(2) ) )$
```

with the resulting plot

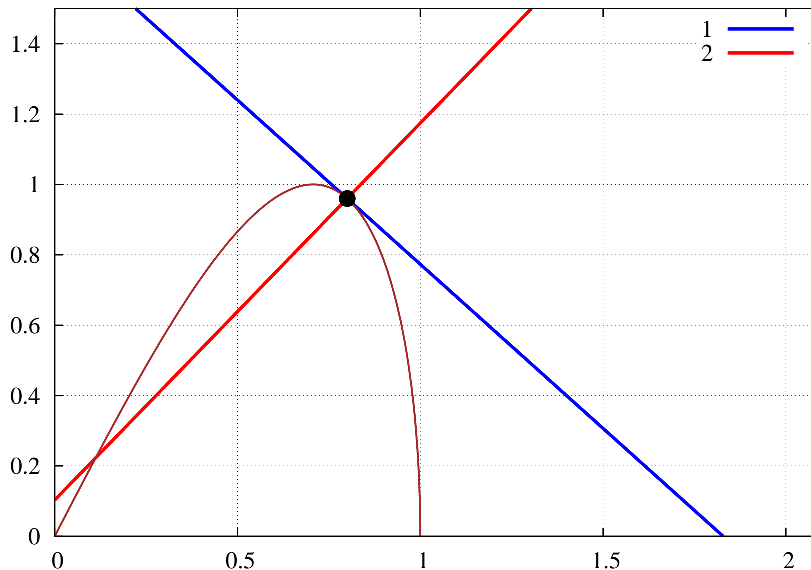

Figure 11: Tangent and Normal to $x = \sin(t)$, $y = \sin(2\,t)$ at point $t = 0.927$ radians

6.5.5 Example 4: Tangent and Normal of a Point on a Polar Plot: $x = r(t)\cos(t), y = r(t)\sin(t)$

We use the polar plot example from ch. 5, in which we took $r(t) = 10/t$, and again t is in radians, and we consider the interval $1 \le t \le \pi$ and find the tangent and normal at the curve point corresponding to $t = 2$ radians. We find the general slope of the tangent by again forming the ratio of the differential dy to the differential dx.

```
(%i32) r : 10/t$
(%i33) xx : r * cos(t)$
(%i34) yy : r * sin(t)$
(%i35) m : diff(yy)/diff(xx)$
(%i36) m : ratsimp(m);
                               sin(t) - t cos(t)
(%o36)                         -----------------
                               t sin(t) + cos(t)
(%i37) m : subst(t = 2.0, m);
(%o37)                             1.2418222
(%i38) mnorm : -1/m;
(%o38)                           - 0.805268
(%i39) x0 : subst(t = 2.0, xx);
(%o39)                           - 2.0807342
(%i40) y0 : subst(t = 2.0, yy);
(%o40)                             4.5464871
(%i41) qdraw(polar(10/t,t,1,3*%pi,lc(brown) ),
           xr(-6.8,10),yr(-3,9),
           ex([y0 + m*(x-x0),y0 + mnorm*(x-x0)],x,-6.8,10 ),
           pts( [ [x0,y0] ], ps(2),pk("t = 2 rad") ) );
```

which produces the plot:

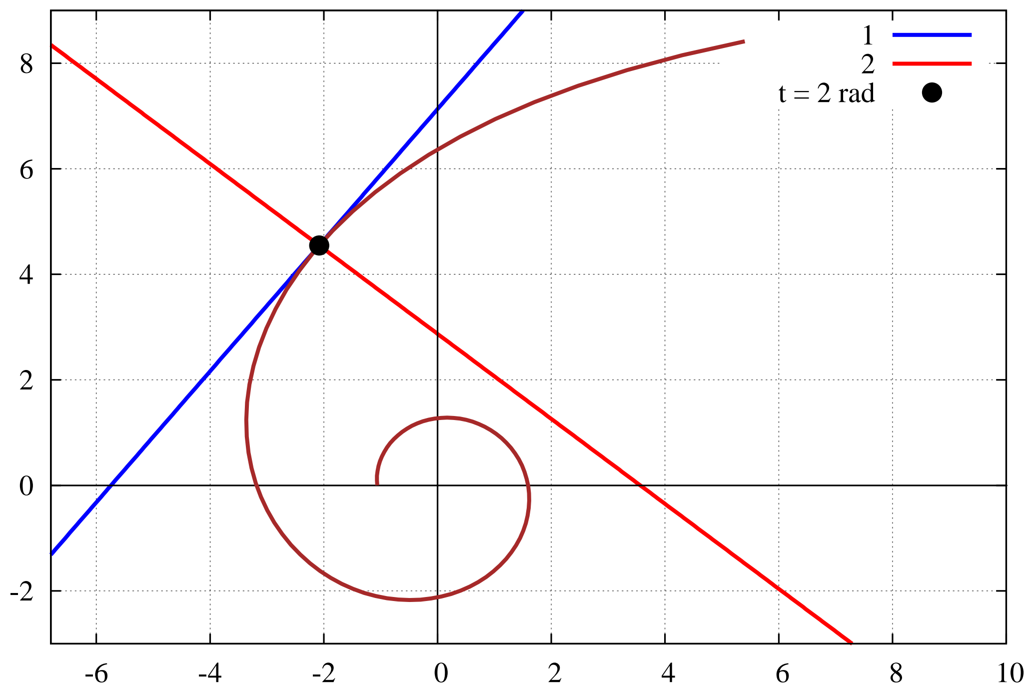

Figure 12: Tangent and Normal to $x = 10\cos(t)/t,\ y = 10\sin(t)/t$ at point $t = 2$ radians

6.6 Limit Examples Using Maxima's limit Function

Maxima has a powerful **limit** function which uses l'Hospital's rule and Taylor series expansions to investigate the limit of a univariate function as the variable approaches some point. The Maxima manual has the following description the Maxima function **limit**:

> Function: **limit(expr, x, val, dir)**
> Function: **limit(expr, x, val)**
> Function: **limit(expr)**
> Computes the limit of **expr** as the real variable **x** approaches the value **val** from the direction **dir**. **dir** may have the value **plus** for a limit from above, **minus** for a limit from below, or may be omitted (implying a two-sided limit is to be computed).
> **limit** uses the following special symbols: **inf** (positive infinity) and **minf** (negative infinity). On output it may also use **und** (undefined), **ind** (indefinite but bounded) and **infinity** (complex infinity).
> **lhospitallim** is the maximum number of times L'Hospital's rule is used in **limit**. This prevents infinite looping in cases like **limit (cot(x)/csc(x), x, 0)**.
> **tlimswitch** when **true** will allow the **limit** command to use Taylor series expansion when necessary.
> **limsubst** prevents **limit** from attempting substitutions on unknown forms. This is to avoid bugs like **limit (f(n)/f(n+1),n,inf)** giving 1. Setting **limsubst** to **true** will allow such substitutions.
> **limit** with one argument is often called upon to simplify constant expressions, for example, **limit (inf-1)**.
> **example (limit)** displays some examples.

Here is the result of calling Maxima's **example** function:

```
(%i1) example(limit)$
(%i2) limit(x*log(x),x,0,plus)
(%o2)                                 0
(%i3) limit((x+1)^(1/x),x,0)
(%o3)                                %e
(%i4) limit(%e^x/x,x,inf)
(%o4)                                inf
(%i5) limit(sin(1/x),x,0)
(%o5)                                ind
```

Most use of **limit** will use the first two ways to call **limit**. The "direction" argument is optional. The default values of the option switches mentioned above are:

```
(%i6) [lhospitallim,tlimswitch,limsubst];
(%o6)                        [4, true, false]
```

Thus the default Maxima behavior is to allow the use of a Taylor series expansion in finding the correct limit. (We will discuss Taylor series expansions soon in this chapter.) The default is also to prevent "substitutions" on unknown (formal) functions. The third (single argument) syntax is illustrated by

```
(%i7) limit(inf - 1);
(%o7)                                inf
```

The expression presented to the **limit** function in input **%i7** contains only known constants, so there are no unbound (formal) parameters like **x** for **limit** to worry about.

Here is a use of **limit** which mimics the calculus definition of a derivative of a power of x.

```
(%i8) limit( ( (x+eps)^3 - x^3 )/eps, eps, 0 );
                                      2
(%o8)                              3 x
```

And a similar use of **limit** with $\ln(x)$:

```
(%i9) limit( (log(x+eps) - log(x))/eps, eps,0 );
                                     1
(%o9)                                -
                                     x
```

What does Maxima do with a typical calculus definition of a derivative of a trigonometric function?

```
(%i10) limit((sin(x+eps)-sin(x))/eps, eps,0 );
Is  sin(x)  positive, negative, or zero?
p;
Is  cos(x)  positive, negative, or zero?
p;
(%o10)                              cos(x)
```

We see above a typical Maxima query before producing an answer. Using **p;** instead of **positive;** is allowed. Likewise one can use **n;** instead of **negative;**.

6.6.1 Discontinuous Functions

A simple example of a discontinuous function can be created using Maxima's **abs** function.
abs(expr) returns either the absolute value **expr**, or (if **expr** is complex) the complex modulus of **expr**.

We first plot the function $|x|/x$.

```
(%i11)  load(draw)$
(%i12)  load(qdraw)$
(%i13) qdraw( yr(-2,2),lw(8),ex(abs(x)/x,x,-1,1 ) )$
```

Here is that plot of $|x|/x$:

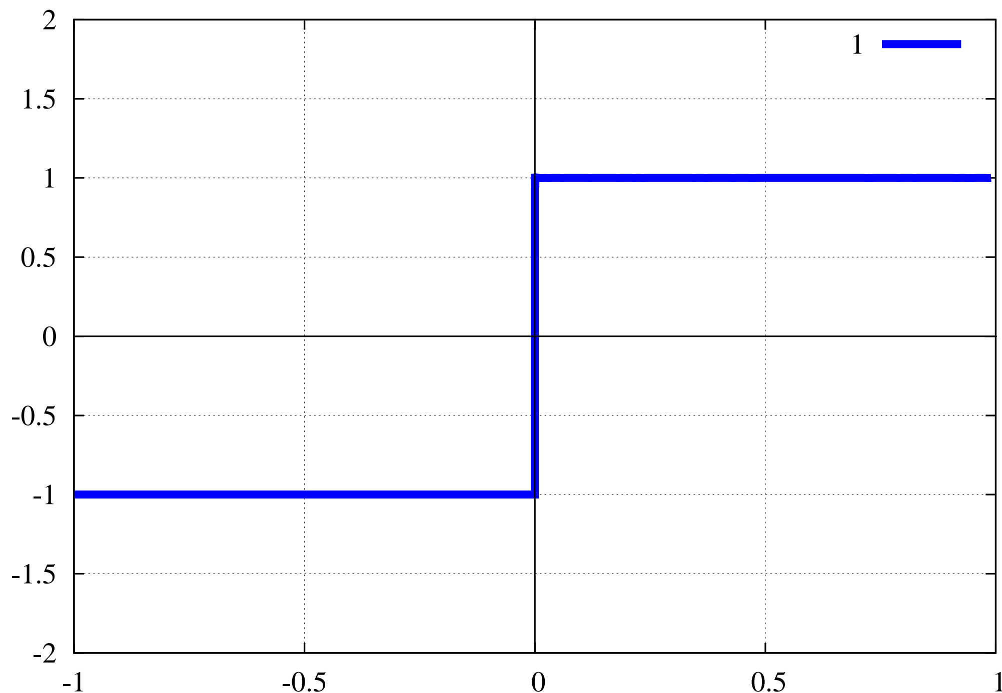

Figure 13: $|x|/x$

Maxima correctly evaluates the one-sided limits:

```
(%i14) limit(abs(x)/x,x,0,plus);
(%o14)                              1
(%i15) limit(abs(x)/x,x,0,minus);
(%o15)                             - 1
(%i16) limit(abs(x)/x,x,0);
(%o16)                             und
```

and Maxima also computes a derivative:

```
(%i17) g : diff(abs(x)/x,x);
                                  1       abs(x)
(%o17)                          ------ - ------
                                abs(x)      2
                                           x
(%i18) g, x = 0.5;
(%o18)                             0.0
(%i19) g, x = - 0.5;
(%o19)                             0.0
(%i20) g,x=0;
Division by 0
 -- an error.  To debug this try debugmode(true);
(%i21) limit(g,x,0,plus);
(%o21)                              0
(%i22) limit(g,x,0,minus);
(%o22)                              0
(%i23) load(vcalc)$
(%i24) plotderiv(abs(x)/x,x,-2,2,-2,2,1)$
                                abs(x)    1       abs(x)
                       plist = [------, ------ - ------]
                                  x     abs(x)      2
                                                   x
```

The derivative does not simplify to 0 since the derivative is undefined at $x = 0$. The plot of the step function and its derivative, as returned by **plotderiv** is

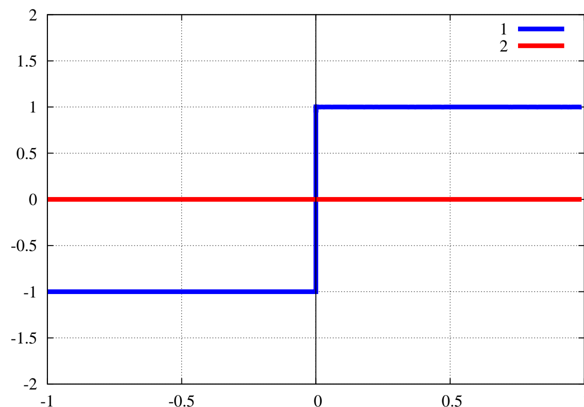

Figure 14: $|x|/x$ and its Maxima derivative

A homemade unit step function can now be defined by adding 1 to lift the function up to the value of 0 for $x < 0$ and then dividing the result by 2 to get a "unit step function".

```
(%i25) mystep : ( (1 + abs(x)/x)/2 , ratsimp(%%) );
                                  abs(x) + x
(%o25)                            ----------
                                     2 x
```

We then use **qdraw** to plot the definition

```
(%i26) qdraw(yr(-1,2),lw(5),ex(mystep,x,-1,1) )$
```

with the result:

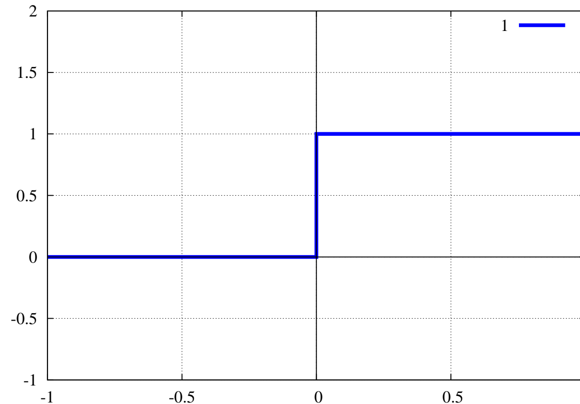

Figure 15: mystep

Maxima has a function called **unit_step** available if you load the **orthopoly** package. This function is "left continuous", since it has the value 0 for $x \leq 1$. However, no derivative is defined, although you can use **gradef**.

```
(%i27) load(orthopoly)$
(%i28) map(unit_step,[-1/10,0,1/10] );
(%o28)                              [0, 0, 1]
(%i29) diff(unit_step(x),x);
                                d
(%o29)                          -- (unit_step(x))
                                dx
(%i30) gradef(unit_step(x),0);
(%o30)                            unit_step(x)
(%i31) diff(unit_step(x),x);
(%o31)                                0
```

Of course, defining the derivative to be 0 everywhere can be dangerous in some circumstances. You can use **unit_step** in a plot using, say, **qdraw(yr(-1,2),lw(5),ex(unit_step(x),x,-1,1));**. Here we use **unit_step** to define a "unit pulse" function **upulse(x,x0,w)** which is a function of x which becomes equal to 1 when $x = x_0$ and has width w.

```
(%i32) upulse(x,x0,w) := unit_step(x-x0) - unit_step(x - (x0+w))$
```

and then make a plot of three black pulses of width 0.5.

```
(%i33) qdraw(yr(-1,2), xr(-3,3),
          ex1( upulse(x,-3,0.5),x,-3,-2.49,lw(5)),
         ex1( upulse(x,-1,0.5),x,-1,-.49,lw(5)),
          ex1( upulse(x,1,0.5),x,1,1.51,lw(5)) )$
```

with the result:

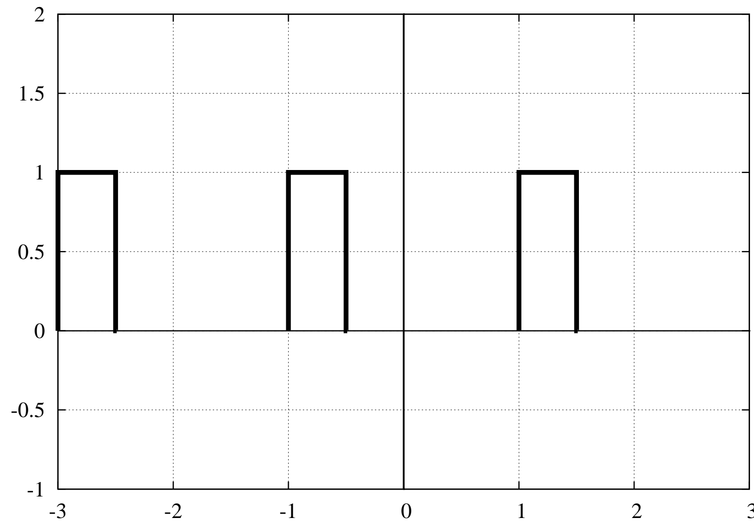

Figure 16: Using upulse(x,x0,w)

Of course, the above drawing can be done more easily with the **poly** function in **qdraw**, which uses **draw2d**'s **polygon** function.

6.6.2 Indefinite Limits

The **example** run showed that $\sin(1/x)$ has no definite limit as the variable x approaches zero. This function oscillates increasingly rapidly between ± 1 as $x \to 0$. It is instructive to make a plot of this function near the origin using the smallest line width:

```
(%i34) qdraw(lw(1),ex(sin(1/x),x,0.001,0.01));
```

The eps image reproduced here actually uses a finer line width than the Windows Gnuplot console window:

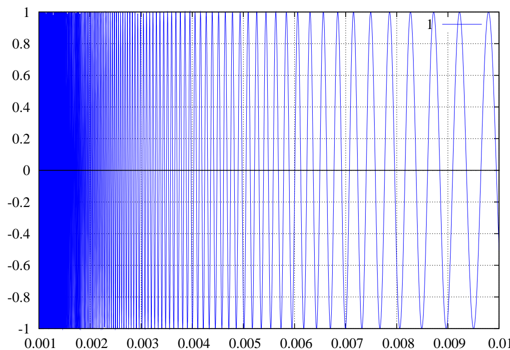

Figure 17: $\sin(1/x)$ Behavior Near $x = 0$

The Maxima **limit** function correctly returns **ind** for "indefinite but bounded" when asked to find the limit as $x \to 0^+$.

```
(%i35) limit(sin(1/x),x,0,plus);
(%o35)                              ind
```

An example of a function which is well behaved at $x = 0$ but whose derivative is indefinite but bounded is $x^2 \sin(1/x)$, which approaches the value 0 at $x = 0$.

```
(%i36) g : x^2*sin(1/x)$
(%i37) limit(g,x,0);
(%o37)                               0
(%i38) dgdx : diff(g,x);
                                  1          1
(%o38)                      2 sin(-) x - cos(-)
                                  x          x
(%i39) limit(dgdx,x,0);
(%o39)                              ind
```

In the first term of the derivative, $x \sin(1/x)$ is driven to 0 by the factor x, but the second term oscillates increasingly rapidly between ± 1 as $x \to 0$. For a plot, we use the smallest line width and color blue for the derivative, and use a thicker red curve for the original function $x^2 \sin(1/x)$.

```
(%i40) qdraw( yr(-1.5,1.5),ex1(2*x*sin(1/x)-cos(1/x),x,-1,1,lw(1),lc(blue)),
              ex1(x^2*sin(1/x),x,-1,1,lc(red) ) )$
```

Here is the plot:

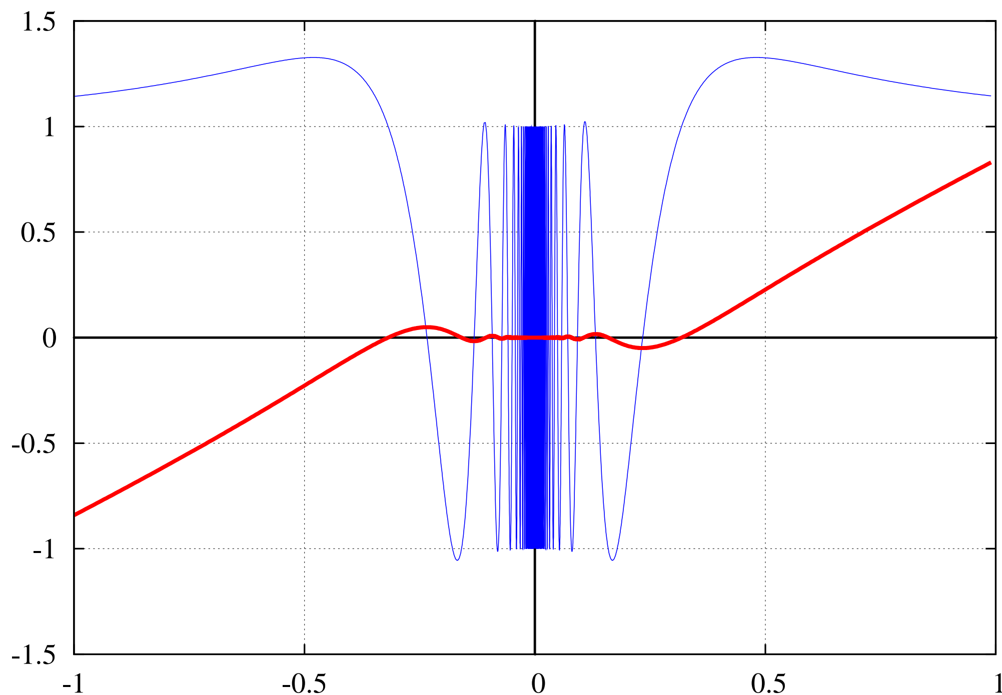

Figure 18: Indefinite but Bounded Behavior of Derivative

6.7 Taylor Series Expansions using taylor

A Taylor (or Laurent) series expansion of a univariate expression has the syntax

taylor(expr, x, a, n)

which will return a Taylor or Laurent series expansion of **expr** in the variable **x** about the point **x = a**, through terms proportional to **(x - a)^n** . Here are some examples of first getting the expansion, then evaluating the expansion at some other point, using both **at** and **subst**.

```
(%i1) t1 : taylor(sqrt(1+x),x,0,5);
                                  2    3      4      5
                             x   x    x    5 x    7 x
(%o1)/T/                 1 + - - -- + -- - ---- + ---- + . . .
                             2   8    16   128    256
(%i2) [ at( t1, x=1 ),subst(x=1, t1 ) ];
                                       365  365
(%o2)                                 [---, ---]
                                       256  256
(%i3) float(%);
(%o3)                         [1.42578125, 1.42578125]
(%i4) t2: taylor (cos(x) - sec(x), x, 0, 5);
                                            4
                                      2    x
(%o4)/T/                         - x  - -- + . . .
                                           6
(%i5) [ at( t2, x=1 ),subst(x=1, t2 ) ];
                                         7    7
(%o5)                                 [- -, - -]
                                         6    6
(%i6) t3 : taylor(cos(x),x,%pi/4,4);
                                      %pi                   %pi 2                    %pi 3
                        sqrt(2)  (x - ---)    sqrt(2)  (x - ---)     sqrt(2)  (x - ---)
          sqrt(2)                      4                     4                        4
(%o6)/T/ ------- - ----------------- - ------------------ + ------------------
             2             2                   4                      12
                                                                         %pi 4
                                                              sqrt(2)  (x - ---)
                                                                            4
                                                            + ------------------ + . . .
                                                                      48
(%i7) ( at(t3, x = %pi/3), factor(%%) );
                        4          3            2
           sqrt(2) (%pi  + 48 %pi  - 1728 %pi  - 41472 %pi + 497664)
(%o7)     ---------------------------------------------------------
                                    995328
```

To expand an expression depending on two variables, say **(x,y)**, there are two essentially different forms. The first form is

taylor(expr, [x,y], [a,b],n)

which will expand in **x** about **x = a** and expand in the variable **y** about **y = b**, up through combined powers of order **n**. If **a = b**, then the simpler syntax:

taylor(expr, [x,y], a, n)

will suffice.

```
(%i8) t4 : taylor(sin(x+y),[x,y],0,3);
                               3         2       2      3
                              x  + 3 y x  + 3 y  x + y
(%o8)/T/            y + x - ------------------------- + . . .
                                        6
(%i9) (subst( [x=%pi/2, y=%pi/2], t4), ratsimp(%%) );
                                      3
                                   %pi  - 6 %pi
(%o9)                            - ------------
                                         6
(%i10) ( at( t4, [x=%pi/2,y=%pi/2]), ratsimp(%%) );
                                      3
                                   %pi  - 6 %pi
(%o10)                           - ------------
                                         6
```

Note the crucial difference between this last example and the next.

```
(%i11) t5 : taylor(sin(x+y),[x,0,3],[y,0,3] );
             3                 2                  3
            y                 y                y   y           2
(%o11)/T/ y - -- + . . . + (1 - -- + . . .) x + (- - + -- + . . .) x
            6                 2                2   12
                                                          2
                                                   1   y           3
                                              + (- - + -- + . . .) x  + . . .
                                                   6   12
(%i12) (subst([x=%pi/2,y=%pi/2],t5),ratsimp(%%) );
                                 5         3
                              %pi  - 32 %pi  + 192 %pi
(%o12)                        ------------------------
                                        192
```

Thus the syntax
taylor(expr, [x,a,nx], [y,b,ny])
will expand to higher combined powers in **x** and **y**.

We can differentiate and integrate a taylor series returned expression:

```
(%i13) t6 : taylor(sin(x),x,0,7 );
                                    3     5      7
                                   x     x      x
(%o13)/T/                     x - -- + --- - ---- + . . .
                                   6    120   5040
(%i14) diff(t6,x);
                                    2     4     6
                                   x     x     x
(%o14)/T/                      1 - -- + -- - --- + . . .
                                   2     24   720
(%i15) integrate(%,x);
                                     7      5      3
                                    x      x      x
(%o15)                            - ---- + --- - -- + x
                                    5040   120    6
(%i16) integrate(t6,x);
                                    8       6      4     2
                                   x       x      x     x
(%o16)                          - ----- + --- - -- + --
                                  40320   720    24    2
```

A common use of the Taylor series is to expand a function of **x** in the form $f(x+dx)$, for dx small.

```
(%i17) taylor(f(x+dx),dx,0,3);
                                                         !
                                             2           !
                                            d            !          2
                                           (---- (f(x + dx))!      ) dx
                                               2         !
                                            ddx          !
                            !                            !dx = 0
                d           !
(%o17)/T/ f(x) + (--- (f(x + dx))!      ) dx + -----------------------------
                ddx         !                                 2
                            !dx = 0
                                                     !
                                          3          !
                                         d           !          3
                                        (---- (f(x + dx))!      ) dx
                                            3        !
                                         ddx         !
                                                     !dx = 0
                                      + ----------------------------- + . . .
                                                    6
(%i18) taylor(cos(x+dx),dx,0,4);
                                            2             3             4
                                   cos(x) dx     sin(x) dx     cos(x) dx
(%o18)/T/ cos(x) - sin(x) dx - ---------- + ---------- + ---------- + . . .
                                        2             6            24
```

Another frequent use of the Taylor series is the expansion of a function which contains a small parameter which we will symbolize by **e**.

```
(%i19) g : log(a/e + sqrt(1+(a/e)^2 ) );
                                          2
                               a         a
(%o19)                     log(- + sqrt(-- + 1))
                               e          2
                                         e
(%i20) taylor(g,e,0,2);
                                                  2
                                                 e
(%o20)/T/          - log(e) + log(2 a) + . . . + ---- + . . .
                                                  2
                                               4 a
```

More examples of Taylor and Laurent series expansions can be found in the Maxima manual.

6.8 Vector Calculus Calculations and Derivations using vcalc.mac

The package **vcalc.mac** has been mentioned in the introduction to this chapter. Here we give a few examples of using this package for vector calculus work.

General Comments: Working with the Batch File vcalcdem.mac

A separate file **vcalcdem.mac** is available with this chapter on the author's webpage. This file is designed to be introduced into Maxima using the command **batch(vcalcdem)** (if **vcalcdem.mac** is in your work directory, say, and you have set up your file search paths as suggested in Ch. 1) or **batch("vcalcdem.mac")**. You should first load in the **vcalc.mac** file which contains the definitions of the vcalc vector calculus functions.

There are three points to emphasize when using batch files in this mode. The first feature is the use of lines like **"this is a comment"$** in the file, which allow you to place readable comments between the usual Maxima input lines. Secondly, a Maxima input which simply defines something and for which there is no need to see the "output" line should end with the dollar sign **$**. Finally, when you "batch in" the file, you see neither the dollar signs, nor the semi-colons which end the other inputs. However, input and output numbers of the form **(%i12)** and **(%o12)** will be

seen for inputs which end with a semi-colon, whereas there are no output numbers for inputs which end with the dollar sign. Once you get a little practice reading the text form of the batch file and comparing that to the Maxima display of the "batched in" file, you should have no problem understanding what is going on. The first lines of **vcalcdem.mac** are:

```
" vcalcdem.mac: sample calculations and derivations"$
" default coordinates are cartesian (x,y,z)"$
" gradient and laplacian of a scalar field "$
depends(f,[x,y,z]);
grad(f);
lap(f);
```

Here is what you will see in your Maxima session:

```
(%i1) load(vcalc);
                vcalc.mac:   for syntax, type: vcalc_syntax();

 CAUTION: global variables set and used in this package:

       hhh1, hhh2, hhh3, uuu1, uuu2, uuu3, nnnsys, nnnprint, tttsimp

(%o1)                          c:/work2/vcalc.mac
(%i2) batch(vcalcdem)$
read and interpret file: #pc:/work5/vcalcdem.mac
(%i3)          vcalcdem.mac: sample calculations and derivations
(%i4)              default coordinates are cartesian (x,y,z)
(%i5)              gradient and laplacian of a scalar field
(%i6)                        depends(f, [x, y, z])
(%o6)                            [f(x, y, z)]
(%i7)                               grad(f)
cartesian  [x, y, z]
                                 df   df   df
(%o7)                           [--, --, --]
                                 dx   dy   dz
(%i8)                              lap(f)
cartesian  [x, y, z]
                                2     2     2
                               d f   d f   d f
(%o8)                          --- + --- + ---
                                 2     2     2
                               dz    dy    dx
```

The default coordinates are cartesian **(x,y,z)**, and by telling Maxima that the otherwise undefined symbol **f** is to be treated as an explicit function of **(x,y,z)**, we get symbolic output from **grad(f)** and **lap(f)** which respectively produce the **gradient** and **laplacian** in the current coordinate system.

For each function used, a reminder is printed to your screen concerning the current coordinate system and the current choice of independent variables.

Three dimensional vectors (the only kind allowed by this package) are represented by lists with three elements. In the default cartesian coordinate system **(x,y,z)**, the first slot is the **x** component of the vector, the second slot is the **y** component, and the third slot is the **z** component.

Now that you have seen the difference between the file **vcalcdem.mac** and the Maxima response when using "batch", we will just show the latter for brevity. You can, of course, look at the file **vcalcdem.mac** with a text editor.

Here the batch file displays the divergence and curl of a general 3-vector in a cartesian coordinate system.

```
(%i9)                  divergence and curl of a vector field
(%i10)                          avec : [ax, ay, az]
(%o10)                             [ax, ay, az]
(%i11)                       depends(avec, [x, y, z])
(%o11)                [ax(x, y, z), ay(x, y, z), az(x, y, z)]
(%i12)                               div(avec)
cartesian  [x, y, z]
                                daz   day   dax
(%o12)                          --- + --- + ---
                                dz    dy    dx
(%i13)                            curl(avec)
cartesian  [x, y, z]
                        daz   day  dax   daz  day   dax
(%o13)                 [--- - ---, --- - ---, --- - ---]
                        dy    dz   dz    dx   dx    dy
```

We next have two vector calculus identities:

```
(%i14)          vector identities true in any coordinate system
(%i15)                           curl(grad(f))
cartesian  [x, y, z]
cartesian  [x, y, z]
(%o15)                             [0, 0, 0]
(%i16)                          div(curl(avec))
cartesian  [x, y, z]
cartesian  [x, y, z]
(%o16)                                 0
```

and an example of finding the Laplacian of a vector field (rather than a scalar field) with an explicit example:

```
(%i17)                       laplacian of a vector field
                                   3       3     2    3
(%i18)                     aa : [x  y, x y  z, x  y z ]
                                 3       3     2    3
(%o18)                        [x  y, x y  z, x  y z ]
(%i19)                               lap(aa)
cartesian  [x, y, z]
                                                3      2
(%o19)                 [6 x y, 6 x y z, 2 y z  + 6 x  y z]
```

and an example of the use of the vector cross product which is included with this package:

```
(%i20)                          vector cross product
(%i21)                           bvec : [bx, by, bz]
(%o21)                              [bx, by, bz]
(%i22)                           lcross(avec, bvec)
(%o22)             [ay bz - az by, az bx - ax bz, ax by - ay bx]
```

We next change the current coordinate system to cylindrical with a choice of the symbols **rho**, **phi**, and **z** as the independent variables. We again start with some general expressions

```
(%i23)              cylindrical coordinates using (rho,phi,z)
(%i24)                      setcoord(cy(rho, phi, z))
(%o24)                                true
(%i25)              gradient and laplacian of a scalar field
(%i26)                      depends(g, [rho, phi, z])
(%o26)                          [g(rho, phi, z)]
```

```
(%i27)                            grad(g)
cylindrical  [rho, phi, z]
                                     dg
                                    ----
                               dg   dphi  dg
(%o27)                       [----, ----, --]
                              drho  rho   dz
(%i28)                            lap(g)
cylindrical  [rho, phi, z]
                                  2
                                 d g
                         dg     -----
                        ----       2     2      2
                        drho    dphi    d g    d g
(%o28)                  ---- + ----- + --- + -----
                        rho       2      2       2
                               rho      dz    drho
(%i29)               divergence and curl of a vector field
(%i30)                      bvec : [brh, bp, bz]
(%o30)                          [brh, bp, bz]
(%i31)                   depends(bvec, [rho, phi, z])
(%o31)       [brh(rho, phi, z), bp(rho, phi, z), bz(rho, phi, z)]
(%i32)                           div(bvec)
cylindrical  [rho, phi, z]
                                 dbp
                                 ----
                          brh    dphi   dbz   dbrh
(%o32)                    --- + ---- + --- + ----
                          rho    rho    dz    drho
(%i33)                          curl(bvec)
cylindrical  [rho, phi, z]
                 dbz                        dbrh
                 ----                       ----
                 dphi   dbp   dbrh   dbz    dphi   bp    dbp
(%o33)          [---- - ---, ---- - ----, - ---- + --- + ----]
                 rho     dz    dz    drho    rho   rho   drho
```

Instead of using **setcoord** to change the current coordinate system, we can insert an extra argument in the vector calculus function we are using. Here is an example of not changing the coordinate system, but telling Maxima to use **r** instead of **rho** with the currently operative cylindrical coordinates.

```
(%i34)   change from cylindrical coordinate label rho to r on the fly:
                                            1
(%i35)                      bvec : [0, -, 0]
                                            r
(%i36)                   div(bvec, cy(r, phi, z))
cylindrical  [r, phi, z]
(%o36)                              0
```

Here is an example of using this method to switch to spherical polar coordinates:

```
(%i37)          change to spherical polar coordinates on the fly
                                      sin(theta)
(%i38)                   cvec : [0, 0, ----------]
                                           2
                                          r
(%i39)                  div(cvec, s(r, theta, phi))
spherical polar  [r, theta, phi]
(%o39)                              0
```

```
(%i40)   coordinate system remains spherical unless explicitly changed
(%i41)                     cvec : [0, 0, r sin(theta)]
(%i42)                               div(cvec)
spherical polar  [r, theta, phi]
(%o42)                                  0
(%i43)          example of div(vec) = 0 everywhere except r = 0
                                       1
(%i44)                          div([--, 0, 0])
                                        2
                                       r
spherical polar  [r, theta, phi]
(%o44)                                  0
```

The best way to get familiar with **vcalc.mac** is just to play with it. Some syntax descriptions are built into the package.

6.9 Maxima Derivation of Vector Calculus Formulas in Cylindrical Coordinates

In this section we start with the cartesian (rectangular) coordinate expressions for the gradient and Laplacian of a scalar expression, and the divergence and curl of a vector expression. We then use Maxima to find the correct forms of these vector calculus formulas in a cylindrical coordinate system.

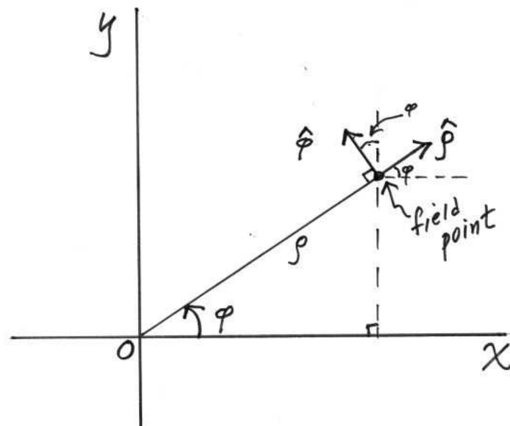

Figure 19: $\boldsymbol{\rho}$ and $\boldsymbol{\phi}$ unit vectors

Consider a change of variable from cartesian coordinates (x, y, z) to cylindrical coordinates (ρ, φ, z). Given (ρ, φ), we obtain (x, y) from the pair of equations $x = \rho \cos \varphi$ and $y = \rho \sin \varphi$. Given (x, y), we obtain (ρ, φ) from the equations $\rho = \sqrt{x^2 + y^2}$ (or $\rho^2 = x^2 + y^2$) and $\tan \varphi = y/x$ (or $\varphi = \arctan(y/x)$).

We consider the derivatives of a general scalar function $f(\rho, \varphi, z)$ which is an **explicit** function of the cylindrical coordinates (ρ, φ, z) and an **implicit** function of the cartesian coordinates (x, y) via the dependence of ρ and φ on x and y.

Likewise we consider a general three dimensional vector $\mathbf{B}(\rho, \varphi, z)$ which is an **explicit** function of ρ, φ, and z and an **implicit function** of (x, y).

Since many of the fundamental equations governing processes in the physical sciences and engineering can be written down in terms of the Laplacian, the divergence, the gradient, and the curl, we concentrate on using Maxima to do the "heavy lifting" (ie., the tedious algebra) to work out formulas for these operations in cylindrical coordinates.

Our approach to the use of vectors here is based on choosing an explicit set of three element lists to represent the cartesian unit vectors. We then use the built-in dot product of lists to implement the scalar product of three-vectors, used to check orthonormality, and we also provide a simple cross product function which can be used to check that the unit vectors we write down are related by the vector cross product to each other in the conventional way (the "right hand rule").

6.9.1 The Calculus Chain Rule in Maxima

Let g be some scalar function which depends **implicitly** on (x, y, z) via an **explicit** dependence on (u, v, w). We want to express the partial derivative of g with respect to x, y, or z in terms of derivatives of g with respect to u, v, and w.

We first tell Maxima to treat g as an **explicit** function of (u, v, w).

```
(%i1) depends(g,[u,v,w]);
(%o1)                             [g(u, v, w)]
```

If, at this point, we ask for the derivative of g with respect to x, we get zero, since Maxima has no information yet about dependence of u, v, and w on x.

```
(%i2) diff(g,x);
(%o2)                                  0
```

We now need to use further **depends** statements, as in

```
(%i3) depends([u,v,w],[x,y,z]);
(%o3)                 [u(x, y, z), v(x, y, z), w(x, y, z)]
```

which now allows Maxima to demonstrate its knowledge of the "calculus chain rule", which Maxima writes as:

```
(%i4) diff(g,x);
                                dg dw   dg dv   dg du
(%o4)                           -- -- + -- -- + -- --
                                dw dx   dv dx   du dx
```

Note how Maxima writes the "chain rule" in this example, which would be written (using partial derivative notation) in a calculus text as

$$\frac{\partial g(u,v,w)}{\partial x} = \frac{\partial g(u,v,w)}{\partial u}\frac{\partial\, u(x,y,z)}{\partial\, x} + \frac{\partial g(u,v,w)}{\partial v}\frac{\partial\, v(x,y,z)}{\partial\, x} + \frac{\partial g(u,v,w)}{\partial w}\frac{\partial\, w(x,y,z)}{\partial\, x} \tag{6.10}$$

In addition to using a **depends** statement to tell Maxima about the (x, y, z) dependence of (u, v, w) , we can use the **gradef** function to replace derivatives with chosen substitutes.

```
(%i5) (gradef(u,x,dudx),gradef(u,y,dudy),gradef(u,z,dudz),
        gradef(v,x,dvdx),gradef(v,y,dvdy),gradef(v,z,dvdz),
         gradef(w,x,dwdx),gradef(w,y,dwdy),gradef(w,z,dwdz) )$
```

Use of **diff** now produces the expression:

```
(%i6) diff(g,x);
                                  dg         dg         dg
(%o6)                        dwdx -- + dvdx -- + dudx --
                                  dw         dv         du
(%i7) grind(%)$
dwdx*'diff(g,w,1)+dvdx*'diff(g,v,1)+dudx*'diff(g,u,1)$
```

This is the method we will use in both this section on cylindrical coordinates and in the next section on spherical polar coordinates.

In the following sections we discuss successive parts of a batch file **cylinder.mac**, available with this chapter on the author's webpage. This file is designed to be introduced into Maxima with the input: **batch(cylinder)** (if **cylinder.mac** is in your work directory, say, and you have set up your file search paths as suggested in Ch. 1), or **batch("c:/work5/cylinder.mac")** if you need to supply the complete path. We have given some orientation concerning batch files in the previous section.

Relating (x, y) to (ρ, φ)

In **cylinder.mac** we use **rh** to represent ρ and **p** to represent the angle φ expressed in radians. The ranges of the independent variables are $0 < \rho < \infty$, $0 \le \varphi \le 2\pi$, and $-\infty \le z \le +\infty$.

Here is the beginning of the batch file **cylinder.mac**. First is defined **c3rule** as a list of replacement "rules" in the form of equations which can be used later by the **subst** function as described by **c3sub**. **rhxy** stands for an expression which produces ρ given (x, y). To get automatic simplification of **rh/abs(rh)** we use the **assume** function.

```
" ------------   cylinder.mac   ------------------------------"$
"  cylindrical coordinates (rho, phi, z ) = (rh, p, z)   "$

" replacement rules x,y,z to rh, p, z  "$

c3rule : [x = rh*cos(p), y = rh*sin(p)  ]$
c3sub(expr) := (subst(c3rule,expr),trigsimp(%%) )$

rhxy : sqrt(x^2 + y^2)$

assume(rh > 0)$
```

which Maxima displays as:

```
(%i1) batch("cylinder.mac")$
read and interpret file: #pc:/work5/cylinder.mac
(%i2)      ------------  cylinder.mac   ----------------------------
(%i3)         cylindrical coordinates (rho, phi, z ) = (rh, p, z)
(%i4)                replacement rules x,y,z to rh, p, z
(%i5)               c3rule : [x = rh cos(p), y = rh sin(p)]
(%i6)         c3sub(expr) := (subst(c3rule, expr), trigsimp(%%))
                                         2    2
(%i7)                        rhxy : sqrt(y  + x )
(%i8)                           assume(rh > 0)
```

Now that you have seen what Maxima does with a file introduced into Maxima via **batch("cylinder.mac")**, we will stop displaying the contents of **cylinder.mac** but just show you Maxima's response. You can look at **cylinder.mac** with a text editor to compare input with output.

We next tell Maxima how to work with the cylindrical coordinates and their derivatives: We let the symbol **drhdx**, for example, hold (for later use) $\partial\,\rho/\partial\,x$.

```
(%i9)          partial derivatives of rho and phi wrt x and y
(%i10)               drhdx : (diff(rhxy, x), c3sub(%%))
(%o10)                             cos(p)
(%i11)               drhdy : (diff(rhxy, y), c3sub(%%))
(%o11)                             sin(p)
                                     y
(%i12)              dpdx : (diff(atan(-), x), c3sub(%%))
                                     x
                                   sin(p)
(%o12)                           - ------
                                     rh
                                     y
(%i13)              dpdy : (diff(atan(-), y), c3sub(%%))
                                     x
                                   cos(p)
(%o13)                             ------
                                     rh
```

We thus have established the derivatives

$$\partial\rho(x,y)/\partial x = \cos\varphi \tag{6.11}$$

$$\partial\rho(x,y)/\partial y = \sin\varphi \tag{6.12}$$

$$\partial\varphi(x,y)/\partial x = -\sin\varphi/\rho \tag{6.13}$$

$$\partial\varphi(x,y)/\partial y = \cos\varphi/\rho \tag{6.14}$$

The batch file has not used any **depends** or **gradef** assignments so far. What the batch file did could have been done **interactively**, starting with the derivative of ρ with respect to x, say, as follows

```
(%i1) rhxy : sqrt(x^2 + y^2);
                                          2    2
(%o1)                          sqrt(y   + x )
(%i2) diff(rhxy,x);
                                          x
(%o2)                          -------------
                                          2    2
                               sqrt(y   + x )
(%i3) subst([x=rh*cos(p),y=rh*sin(p)],%);
                                   cos(p) rh
(%o3)                   -------------------------------
                                2      2      2      2
                        sqrt(sin  (p) rh  + cos  (p) rh )
(%i4) trigsimp(%);
                                   cos(p) rh
(%o4)                              ---------
                                    abs(rh)
(%i5) assume(rh > 0)$
(%i6) ev (%o4);
(%o6)                               cos(p)
```

and if we had used **assume(rh > 0)** earlier, we would not have had to include the **ev** step later, as you can verify by restarting Maxima or using **kill(all)** to redo the calculation.

Returning to the batch file, the necessary variable dependencies and the desired derivative replacements are assigned:

```
(%i14)              tell Maxima rh=rh(x,y) and p = p(x,y)
(%i15)          (gradef(rh, x, drhdx), gradef(rh, y, drhdy))
(%i16)            (gradef(p, x, dpdx), gradef(p, y, dpdy))
(%i17)                    depends([rh, p], [x, y])
```

6.9.2 Laplacian $\nabla^2 f(\rho,\varphi,z)$

The Laplacian of a scalar function f depending on (x,y,z),is defined as

$$\nabla^2 f(x,y,z) \equiv \frac{\partial^2 f(x,y,z)}{\partial x^2} + \frac{\partial^2 f(x,y,z)}{\partial y^2} + \frac{\partial^2 f(x,y,z)}{\partial z^2} \tag{6.15}$$

The batch file now calculates the Laplacian of a scalar function f when the function depends **explicitly** on the cylindrical coordinates (ρ,φ,z) and hence depends **implicitly** on (x,y) since ρ and φ each depend on (x,y). This latter dependence has already been introduced via **depends**, together with an automatic derivative substitution via **gradef**.

The batch file tells Maxima to treat **f** as an explicit function of **rh**, **p**, and **z** via **depends(f,[rh,p,z])**. At that point, if the batch file had asked for the first derivative of **f** with respect to **x**, the response would be

```
(%i98) diff (f,x);
                                        df
                                        -- sin(p)
                          df            dp
(%o98)                    --- cos(p) - ---------
                          drh              rh
```

The cartesian form of the Laplacian of f can thus be expressed in terms of the fundamental set of derivatives $\partial \rho / \partial x$, etc., which have already been established above. The next section of **cylinder.mac** then produces the response:

```
(%i18)         -----------------------------------------------
(%i19)           Laplacian of a scalar function f
(%i20)         -----------------------------------------------
(%i21)           tell Maxima to treat scalar function f as an
(%i22)                    explicit function of (rh,p,z)
(%i23)                      depends(f, [rh, p, z])
(%o23)                           [f(rh, p, z)]
(%i24)      calculate the Laplacian of the scalar function f(rh,p,z)
(%i25)                   using the cartesian definition
(%i26) (diff(f, z, 2) + diff(f, y, 2) + diff(f, x, 2), trigsimp(%%),
                                                           multthru(%%))
                                   2
                                  d f
                          df      ---
                          ---       2     2      2
                          drh     dp     d f    d f
(%o26)                    --- + --- + --- + ----
                          rh        2      2      2
                                  rh     dz     drh
(%i27)                             grind(%)
'diff(f,rh,1)/rh+'diff(f,p,2)/rh^2+'diff(f,z,2)+'diff(f,rh,2)$
```

We then have the result:

$$\nabla^2 f(\rho, \varphi, z) = \frac{1}{\rho}\frac{\partial f}{\partial \rho} + \frac{\partial^2 f}{\partial \rho}^2 + \frac{1}{\rho^2}\frac{\partial^2 f}{\partial \varphi^2} + \frac{\partial^2 f}{\partial z^2} \tag{6.16}$$

The two terms involving derivatives with respect to ρ can be combined:

$$\frac{1}{\rho}\frac{\partial f}{\partial \rho} + \frac{\partial^2 f}{\partial \rho^2} = \frac{1}{\rho}\frac{\partial}{\partial \rho}\left(\rho \frac{\partial f}{\partial \rho}\right) \tag{6.17}$$

We clearly need to avoid points for which $\rho = 0$.

It is one thing to use Maxima to help derive the correct form of an operation like the Laplacian operator in cylindrical coordinates, and another to build a usable function which can be used (without repeating a derivation each time!) to calculate the Laplacian when we have some concrete function we are interested in.

The file **vcalc.mac**, available with this chapter on the author's webpage, contains usable functions for the Laplacian, gradient, divergence, and curl for the three coordinate systems: cartesian, cylindrical, and spherical polar.

However, here is a Maxima function specifically designed only for cylindrical coordinates which will calculate the Laplacian of a scalar expression in cylindrical coordinates. No effort at simplification is made inside this function; the result can be massaged by the alert user which Maxima always assumes is available. (Note: the Laplacian and other functions in **vcalc.mac** start with the general expressions valid in any orthonormal coordinate system, using the appropriate "scale factors" (h1,h2,h3), and some simplification is done. Also, the laplacian in **vcalc.mac** will correctly compute the Laplacian of a vector field as well as a scalar field.)

```
(%i1) cylaplacian(expr,rho,phi,z) :=
            (diff(expr,rho)/rho + diff(expr,phi,2)/rho^2 +
               diff(expr,rho,2) + diff(expr,z,2))$
```

We can show that the combinations $\rho^n \cos(n\,\varphi)$ and $\rho^n\ \sin(n\,\varphi)$ are each solutions of Laplace's equation $\nabla^2 \mathbf{u} = 0$.

```
(%i2) ( cylaplacian(rh^n*cos(n*p),rh,p,z), factor(%%) );
(%o2)                              0
(%i3) ( cylaplacian(rh^n*sin(n*p),rh,p,z), factor(%%) );
(%o3)                              0
(%i4) ( cylaplacian(rh^(-n)*cos(n*p),rh,p,z), factor(%%) );
(%o4)                              0
(%i5) ( cylaplacian(rh^(-n)*sin(n*p),rh,p,z), factor(%%) );
(%o5)                              0
```

Another general solution is $\ln(\rho)$:

```
(%i6) cylaplacian(log(rh),rh,p,z);
(%o6)                              0
```

Here is another example of the use of this Maxima function. The expression $\mathbf{u} = (2/3)\,(\rho - 1/\rho)\sin(\varphi)$ is proposed as the solution to a problem defined by: (partial differential equation: pde) $\nabla^2 \mathbf{u} = 0$ for $1 \le \rho \le 2$, and (boundary conditions: bc) $\mathbf{u}(1, \varphi) = 0$, and $\mathbf{u}(2, \varphi) = \sin(\varphi)$. Here we use Maxima to check the three required solution properties.

```
(%i7) u : 2*(rh - 1/rh)*sin(p)/3$
(%i8) (cylaplacian(u,rh,p,z), ratsimp(%%) );
(%o8)                              0
(%i9) [subst(rh=1,u),subst(rh=2,u) ];
(%o9)                         [0, sin(p)]
```

6.9.3 Gradient $\nabla f(\rho, \varphi, z)$

There are several ways one can work with vector calculus problems. One method (not used here) is to use symbols for a set of orthogonal unit vectors, and assume a set of properties for these unit vectors without connecting the set to any specific list basis representation. One can then construct the dot product, the cross product and the derivative operations in terms of the coefficients of these symbolic unit vectors (using **ratcoeff**, for example).

Alternatively (and used here), one can define 3-vectors in terms of three element lists, in which the first element of the list contains the $\mathbf{x}$ axis component of the vector, the second element of the list contains the $\mathbf{y}$ axis component, and the third element contains the $\mathbf{z}$ component.

This second method is less abstract and closer to the beginning student's experience of vectors, and provides a straightforward path which can be used with any orthonormal coordinate system.

The unit vector along the $\mathbf{x}$ axis ($\hat{\mathbf{x}}$) is represented by **xu = [1,0,0]** (for "x - unit vector"), the unit vector along the $\mathbf{y}$ axis ($\hat{\mathbf{y}}$) is represented by **yu = [0,1,0]**, and the unit vector along the $\mathbf{z}$ axis ($\hat{\mathbf{z}}$) is represented by **zu = [0,0,1]**.

Let's return to the batch file **cylinder.mac**, where we define **lcross(u,v)** to calculate the vector cross product $\mathbf{u} \times \mathbf{v}$ when we are using three element lists to represent vectors.

Note that our "orthonormality checks" use the built-in "dot product" of lists provided by the period **.** (which is also used for non-commutative matrix multiplication). The dot product of two vectors represented by Maxima lists is obtained by placing a period between the lists. We have checked that the cartesian vectors defined are "unit vectors" (the dot product yields unity) and are also mutually orthogonal (ie., at right angles to each other) which is equivalent to the dot product of a pair of different unit vectors being zero.

This section of **cylinder.mac** begins with the code for **lcross** which looks like:

```
lcross(u,v)  := (
          ( u[2]*v[3] - u[3]*v[2] )*xu +
           ( u[3]*v[1] - u[1]*v[3]  )*yu +
           ( u[1]*v[2] - u[2]*v[1] )*zu )$
```

Note, in the batch file output, using **display2d:true** (the default), how the list element numbers are displayed as subscripts.

```
(%i28)          ---------------------------------------------
(%i29)                      Unit Vectors
(%i30)          ---------------------------------------------
(%i31)         cross product rule when using lists for vectors
(%i32) lcross(u, v) := (u  v  - u  v ) zu + (u  v  - u  v ) yu
                          1  2    2  1        3  1    1  3
                                                    + (u  v  - u  v ) xu
                                                        2  3    3  2
(%i33)            cross product of parallel vectors is zero
(%i34)               lcross([a, b, c], [n a, n b, n c])
(%o34)                                0
(%i35)                 a function we can use with map
(%i36)                 apcr(ll) := apply('lcross, ll)
(%i37)              3d cartesian unit vectors using lists
(%i38)        (xu : [1, 0, 0], yu : [0, 1, 0], zu : [0, 0, 1])
(%i39)         orthonormality checks on cartesian unit vectors
(%i40)     [xu . xu, yu . yu, zu . zu, xu . yu, xu . zu, yu . zu]
(%o40)                      [1, 1, 1, 0, 0, 0]
(%i41)    low tech check of cross products of cartesian unit vectors
(%i42)                      lcross(xu, yu) - zu
(%o42)                           [0, 0, 0]
(%i43)                      lcross(yu, zu) - xu
(%o43)                           [0, 0, 0]
(%i44)                      lcross(zu, xu) - yu
(%o44)                           [0, 0, 0]
(%i45)        [lcross(xu, xu), lcross(yu, yu), lcross(zu, zu)]
(%o45)                [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
(%i46)   high tech checks of cross products of cartesian unit vectors
(%i47)    map('apcr, [[xu, yu], [yu, zu], [zu, xu]]) - [zu, xu, yu]
(%o47)                [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
(%i48)           map('apcr, [[xu, xu], [yu, yu], [zu, zu]])
(%o48)                [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Thus we have cartesian unit vector relations such as $\hat{\mathbf{x}} \cdot \hat{\mathbf{x}} = 1$, and $\hat{\mathbf{x}} \cdot \hat{\mathbf{y}} = 0$, and $\hat{\mathbf{x}} \times \hat{\mathbf{y}} = \hat{\mathbf{z}}$.

The unit vector $\hat{\boldsymbol{\rho}}$ along the direction of increasing ρ at the point (ρ, φ, z) is defined in terms of the cartesian unit vectors $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ via

$$\hat{\boldsymbol{\rho}} = \hat{\mathbf{x}} \cos\varphi + \hat{\mathbf{y}} \sin\varphi \tag{6.18}$$

Because the direction of $\hat{\boldsymbol{\rho}}$ depends on φ, a more explicit notation would be $\hat{\boldsymbol{\rho}}(\varphi)$, but following convention, we surpress that dependence in the following.

The unit vector $\hat{\boldsymbol{\varphi}}$ along the direction of increasing φ at the point (ρ, φ, z) is

$$\hat{\boldsymbol{\varphi}} = -\hat{\mathbf{x}} \sin\varphi + \hat{\mathbf{y}} \cos\varphi \tag{6.19}$$

Again, because the direction of $\hat{\boldsymbol{\varphi}}$ depends on φ, a more explicit notation would be $\hat{\boldsymbol{\varphi}}(\varphi)$, but following conventional use we surpress that dependence in the following. We use the symbol **rhu** ("rh-unit-vec") for $\hat{\boldsymbol{\rho}}$, and the symbol **pu** for $\hat{\boldsymbol{\varphi}}$.

```
(%i49)       cylindrical coordinate unit vectors rho-hat, phi-hat
(%i50)                    rhu : sin(p) yu + cos(p) xu
(%o50)                        [cos(p), sin(p), 0]
(%i51)                    pu : cos(p) yu - sin(p) xu
(%o51)                       [- sin(p), cos(p), 0]
(%i52)               orthonormality checks on unit vectors
(%i53) ([rhu . rhu, pu . pu, zu . zu, rhu . pu, rhu . zu, pu . zu],
                                                            trigsimp(%%))
(%o53)                        [1, 1, 1, 0, 0, 0]
(%i54)                 low tech check of cross products
(%i55)               (lcross(rhu, pu), trigsimp(%%)) - zu
(%o55)                             [0, 0, 0]
(%i56)               (lcross(pu, zu), trigsimp(%%)) - rhu
(%o56)                             [0, 0, 0]
(%i57)               (lcross(zu, rhu), trigsimp(%%)) - pu
(%o57)                             [0, 0, 0]
(%i58)                [lcross(rhu, rhu), lcross(pu, pu)]
(%o58)                      [[0, 0, 0], [0, 0, 0]]
(%i59)                high tech checks of cross products
(%i60) (map('apcr, [[rhu, pu], [pu, zu], [zu, rhu]]), trigsimp(%%))
                                                          - [zu, rhu, pu]
(%o60)                 [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
(%i61)                map('apcr, [[rhu, rhu], [pu, pu]])
(%o61)                      [[0, 0, 0], [0, 0, 0]]
```

Thus we have cylindrical unit vector relations such as $\hat{\boldsymbol{\rho}} \cdot \hat{\boldsymbol{\rho}} = 1$, $\hat{\boldsymbol{\rho}} \cdot \hat{\boldsymbol{\varphi}} = 0$, and $\hat{\boldsymbol{\rho}} \times \hat{\boldsymbol{\varphi}} = \hat{\mathbf{z}}$.

The gradient of a scalar function $f(x,y,z)$ is defined by

$$\boldsymbol{\nabla} f(x,y,z) = \hat{\mathbf{x}}\,\frac{\partial f}{\partial x} + \hat{\mathbf{y}}\,\frac{\partial f}{\partial y} + \hat{\mathbf{z}}\,\frac{\partial f}{\partial z} \tag{6.20}$$

For a scalar function f which depends explicitly on (ρ,φ,z) and hence implicitly on (x,y), we can use the relations already established in **cylinder.mac** to convert the cartesian expression of the gradient of f (called here **fgradient**) into an expression in terms of derivatives with respect to (ρ,φ,z).

Returning to the output of **cylinder.mac**:

```
(%i62)         --------------------------------------------
(%i63)                 Gradient of a Scalar Function f
(%i64)         --------------------------------------------
(%i65)              cartesian def. of gradient of scalar f
(%i66)     fgradient : diff(f, z) zu + diff(f, y) yu + diff(f, x) xu
```

The $\boldsymbol{\rho}$ component of a vector $\mathbf{A}$ (at the point $(\boldsymbol{\rho},\boldsymbol{\varphi},\mathbf{z})$) is given by

$$A_\rho = \hat{\boldsymbol{\rho}} \cdot \mathbf{A}, \tag{6.21}$$

and the $\boldsymbol{\varphi}$ component of a vector $\mathbf{A}$ at the point $(\boldsymbol{\rho},\boldsymbol{\varphi},\mathbf{z})$ is given by

$$A_\varphi = \hat{\boldsymbol{\varphi}} \cdot \mathbf{A}. \tag{6.22}$$

Our batch file follows this pattern to get the (ρ,φ,z) components of the vector $\boldsymbol{\nabla} f$:

```
(%i67)                rho, phi, and z components of grad(f)
(%i68)          fgradient_rh : (rhu . fgradient, trigsimp(%%))
                                            df
(%o68)                                      ---
                                            drh
```

```
(%i69)          fgradient_p : (pu . fgradient, trigsimp(%%))
                                    df
                                    --
                                    dp
(%o69)                              --
                                    rh
(%i70)          fgradient_z : (zu . fgradient, trigsimp(%%))
                                    df
(%o70)                              --
                                    dz
```

Hence we have derived

$$\boldsymbol{\nabla} f(\rho,\varphi,z) = \hat{\boldsymbol{\rho}}\,\frac{\partial f}{\partial \rho} + \hat{\boldsymbol{\varphi}}\,\frac{1}{\rho}\,\frac{\partial f}{\partial \varphi} + \hat{\mathbf{z}}\,\frac{\partial f}{\partial z}. \tag{6.23}$$

6.9.4 Divergence $\boldsymbol{\nabla}\cdot\mathbf{B}(\rho,\varphi,z)$

In cartesian coordinates the divergence of a three dimensional vector field $\mathbf{B}(x,y,z)$ can be calculated with the equation

$$\boldsymbol{\nabla}\cdot\mathbf{B}(x,y,z) = \frac{\partial B_x}{\partial x} + \frac{\partial B_y}{\partial y} + \frac{\partial B_z}{\partial z} \tag{6.24}$$

Consider a vector field $\mathbf{B}(\rho,\varphi,z)$ which is an explicit function of the cylindrical coordinates $\rho,\varphi,z)$ and hence an implicit function of (x,y). We will use the symbol **bvec** to represent $\mathbf{B}(\rho,\varphi,z)$, and use the symbols **bx**, **by**, and **bz** to represent the x, y, and z components of $\mathbf{B}(\rho,\varphi,z)$.

The ρ component (the component in the direction of increasing ρ with constant φ and constant z) of $\mathbf{B}(\rho,\varphi,z)$ at the point (ρ,φ,z) is given by

$$B_\rho(\rho,\varphi,z) = \hat{\boldsymbol{\rho}}\cdot\mathbf{B}(\rho,\varphi,z). \tag{6.25}$$

We use the symbol **brh** for $B_\rho(\rho,\varphi,z)$. The component of $\mathbf{B}(\rho,\varphi,z)$ at the point (ρ,φ,z) in the direction of increasing φ (with constant ρ and constant z) is given by the equation

$$B_\varphi(\rho,\varphi,z) = \hat{\boldsymbol{\varphi}}\cdot\mathbf{B}(\rho,\varphi,z). \tag{6.26}$$

We use the symbol **bp** for $B_\varphi(\rho,\varphi,z)$. Returning to the output of **cylinder.mac** batch file:

```
(%i71)          -----------------------------------------------
(%i72)                Divergence of a Vector bvec
(%i73)          -----------------------------------------------
(%i74)                   bvec : bz zu + by yu + bx xu
(%o74)                           [bx, by, bz]
(%i75)         two equations which relate cylindrical components
(%i76)                 of bvec to the cartesian components
(%i77)                      eq1 : brh = rhu . bvec
(%o77)                    brh = by sin(p) + bx cos(p)
(%i78)                       eq2 : bp = pu . bvec
(%o78)                    bp = by cos(p) - bx sin(p)
(%i79)                      invert these equations
(%i80)       sol : (linsolve([eq1, eq2], [bx, by]), trigsimp(%%))
(%o80)     [bx = brh cos(p) - bp sin(p), by = brh sin(p) + bp cos(p)]
(%i81)                     [bx, by] : map('rhs, sol)
(%i82)          tell Maxima to treat cylindrical components as
(%i83)                   explicit functions of (rh,p,z)
(%i84)                depends([brh, bp, bz], [rh, p, z])
(%o84)            [brh(rh, p, z), bp(rh, p, z), bz(rh, p, z)]
(%i85)                 calculate the divergence of bvec
(%i86) bdivergence : (diff(bz, z) + diff(by, y) + diff(bx, x), trigsimp(%%),
                                                            multthru(%%))
                                  dbp
                                  ---
                            brh   dp    dbz   dbrh
(%o86)                      --- + --- + --- + ----
                            rh    rh    dz    drh
```

Hence we have derived the result:

$$\nabla \cdot \mathbf{B}(\rho,\varphi,z) = \frac{1}{\rho}\frac{\partial}{\partial\,\rho}\,(\rho\,B_\rho) + \frac{1}{\rho}\frac{\partial}{\partial\,\varphi}\,B_\varphi + \frac{\partial\,B_z}{\partial\,z}, \tag{6.27}$$

in which we have used the identity

$$\frac{1}{\rho}\frac{\partial}{\partial\,\rho}\,(\rho\,B_\rho) = \frac{B_\rho}{\rho} + \frac{\partial}{\partial\,\rho}\,B_\rho \tag{6.28}$$

6.9.5 Curl $\nabla \times \mathbf{B}(\rho,\varphi,z)$

In cartesian coordinates the curl of a three dimensional vector field $\mathbf{B}(x,y,z)$ can be calculated with the equation

$$\nabla \times \mathbf{B}(x,y,z) = \hat{\mathbf{x}}\left(\frac{\partial B_z}{\partial y} - \frac{\partial B_y}{\partial z}\right) + \hat{\mathbf{y}}\left(\frac{\partial B_x}{\partial z} - \frac{\partial B_z}{\partial x}\right) + \hat{\mathbf{z}}\left(\frac{\partial B_y}{\partial x} - \frac{\partial B_x}{\partial y}\right) \tag{6.29}$$

Remember that we have already bound the symbols **bx** and **by** to linear combinations of **brh** and **bp**, and have told Maxima (using **depends**) to treat **brh** and **bp** as explicit functions of **(rh,p,z)**. Hence our assignment of the symbol **bcurl** in **cylinder.mac** will result in Maxima using the calculus chain rule. We can then extract the **cylindrical components** of $\mathbf{B}$ by taking the dot product of the cylindrical unit vectors with $\mathbf{B}$, just as we did to get the cylindrical components of ∇f.

Returning to the output of **cylinder.mac**:

```
(%i87)     -------------------------------------------------------
(%i88)          Cylindrical Components of Curl(bvec)
(%i89)     -------------------------------------------------------
(%i90)                cartesian definition of curl(vec)
(%i91) bcurl : (diff(by, x) - diff(bx, y)) zu + (diff(bx, z) - diff(bz, x)) yu
                                              + (diff(bz, y) - diff(by, z)) xu
(%i92)             find cylindrical components of curl(bvec)
(%i93)       bcurl_rh : (rhu . bcurl, trigsimp(%%), multthru(%%))
                                 dbz
                                 ---
                                 dp     dbp
(%o93)                           --- - ---
                                 rh     dz
(%i94)        bcurl_p : (pu . bcurl, trigsimp(%%), multthru(%%))
                                dbrh   dbz
(%o94)                          ---- - ---
                                 dz    drh
(%i95)        bcurl_z : (zu . bcurl, trigsimp(%%), multthru(%%))
                                dbrh
                                ----
                                 dp     bp    dbp
(%o95)                        - ---- + -- + ---
                                 rh     rh    drh
(%i96)      -------------------------------------------------------
```

Hence we have derived

$$\nabla \times \mathbf{B}(\rho,\varphi,z) = \hat{\boldsymbol{\rho}}\left(\frac{1}{\rho}\frac{\partial\,B_z}{\partial\,\varphi} - \frac{\partial\,B_\varphi}{\partial\,z}\right) + \hat{\boldsymbol{\varphi}}\left(\frac{\partial\,B_\rho}{\partial\,z} - \frac{\partial\,B_z}{\partial\,\rho}\right) + \hat{\mathbf{z}}\left(\frac{1}{\rho}\frac{\partial}{\partial\,\rho}\,(\rho\,B_\varphi) - \frac{1}{\rho}\frac{\partial\,B_\rho}{\partial\,\varphi}\right) \tag{6.30}$$

in which we have used

$$\frac{B_\varphi}{\rho} + \frac{\partial\,B_\phi}{\partial\,\rho} = \frac{1}{\rho}\frac{\partial}{\partial\,\rho}\,(\rho\,B_\varphi). \tag{6.31}$$

6.10 Maxima Derivation of Vector Calculus Formulas in Spherical Polar Coordinates

The batch file **sphere.mac**, available on the author's webpage with this chapter, uses Maxima to apply the **same method** we used in the previous section, but here we derive **spherical polar coordinate expressions** for the gradient, divergence, curl and Laplacian. We will include less commentary in this section since the general approach is the same.

We consider a change of variable from cartesian coordinates (x, y, z) to spherical polar coordinates (r, θ, φ). Given (r, θ, φ), we obtain (x, y, z) from the equations $x = r \sin\theta \cos\varphi$, $y = r \sin\theta \sin\varphi$, and $z = r \cos\theta$. Given (x, y, z), we obtain (r, θ, φ) from the equations $r = \sqrt{x^2 + y^2 + z^2}$, $\cos\theta = z/\sqrt{x^2 + y^2 + z^2}$, and $\tan\varphi = y/x$.

In **sphere.mac** we use **t** to represent the angle θ and **p** to represent the angle φ (both expressed in radians). The ranges of the independent variables are $0 < r < \infty$, $0 < \theta < \pi$, and $0 \leq \varphi < 2\pi$.

We first define **s3rule** as a list of replacement "rules" in the form of equations which can be used later by the **subst** function employed by **s3sub**. To get automatic simplification of **r/abs(r)** and **sin(t)/abs(sin(t)** we use the **assume** function.

Here is the beginning of the **sphere.mac** batch file output when used with Maxima ver. 5.21.1:

```
(%i1) batch("sphere.mac")$
read and interpret file: #pc:/work5/sphere.mac
(%i2)        ------------  sphere.mac   ----------------------------
(%i3)         spherical polar coordinates (r,theta,phi ) = (r,t,p)
(%i4)                   replacement rules x,y,z to r,t,p
(%i5)  s3rule : [x = r sin(t) cos(p), y = r sin(t) sin(p), z = r cos(t)]
(%i6)         s3sub(expr) := (subst(s3rule, expr), trigsimp(%%))
(%i7)                       assume(r > 0, sin(t) > 0)
                                         2    2    2
(%i8)                       rxyz : sqrt(z  + y  + x )
(%i9)      partial derivatives of r, theta, and phi wrt x, y, and z
(%i10)                  drdx : (diff(rxyz, x), s3sub(%%))
(%o10)                            cos(p) sin(t)
(%i11)                  drdy : (diff(rxyz, y), s3sub(%%))
(%o11)                            sin(p) sin(t)
(%i12)                  drdz : (diff(rxyz, z), s3sub(%%))
(%o12)                               cos(t)
                                       z
(%i13)               dtdx : (diff(acos(----), x), s3sub(%%))
                                      rxyz
                                cos(p) cos(t)
(%o13)                          -------------
                                       r
                                       z
(%i14)               dtdy : (diff(acos(----), y), s3sub(%%))
                                      rxyz
                                sin(p) cos(t)
(%o14)                          -------------
                                       r
                                       z
(%i15)               dtdz : (diff(acos(----), z), s3sub(%%))
                                      rxyz
                                    sin(t)
(%o15)                            - ------
                                       r
                                       y
```

```
(%i16)             dpdx : (diff(atan(-), x), s3sub(%%))
                                      x
                                    sin(p)
(%o16)                           - --------
                                   r sin(t)
                                      y
(%i17)             dpdy : (diff(atan(-), y), s3sub(%%))
                                      x
                                   cos(p)
(%o17)                            --------
                                  r sin(t)
(%i18)             tell Maxima r=r(x,y,z), t = t(x,y,z),
(%i19)                        and p = p(x,y)
(%i20)    (gradef(r, x, drdx), gradef(r, y, drdy), gradef(r, z, drdz))
(%i21)    (gradef(t, x, dtdx), gradef(t, y, dtdy), gradef(t, z, dtdz))
(%i22)            (gradef(p, x, dpdx), gradef(p, y, dpdy))
(%i23)                   depends([r, t], [x, y, z])
(%i24)                       depends(p, [x, y])
```

The batch file next calculates the Laplacian of a scalar function f.

```
(%i25)        ------------------------------------------------
(%i26)          Laplacian of a scalar function f
(%i27)        ------------------------------------------------
(%i28)          tell Maxima to treat scalar function f as an
(%i29)                    explicit function of (r,t,p)
(%i30)                      depends(f, [r, t, p])
(%o30)                          [f(r, t, p)]
(%i31)     calculate the Laplacian of the scalar function f(r,t,p)
(%i32)                  using the cartesian definition
(%i33) (diff(f, z, 2) + diff(f, y, 2) + diff(f, x, 2), trigsimp(%%),
                                                  scanmap('multthru, %%))
                                   2                2
                                  d f              d f
                  df              ---         df   ---
                  -- cos(t)         2       2 --     2    2
                  dt              dp          dr   dt    d f
(%o33)            --------- + ---------- + ---- + --- + ---
                   2            2    2        r     2      2
                  r  sin(t)    r  sin (t)          r     dr
(%i34)                             grind(%)
'diff(f,t,1)*cos(t)/(r^2*sin(t))+'diff(f,p,2)/(r^2*sin(t)^2)+2*'diff(f,r,1)/r
                                        +'diff(f,t,2)/r^2+'diff(f,r,2)$
```

Hence the form of the Laplacian of a scalar field in spherical polar coordinates:

$$\nabla^2 f(r,\theta,\varphi) = \frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial f}{\partial r}\right) + \frac{1}{r^2\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial f}{\partial\theta}\right) + \frac{1}{r^2\sin^2\theta}\frac{\partial^2 f}{\partial\varphi^2} \tag{6.32}$$

in which we have used the identities

$$\frac{\partial^2 f}{\partial r^2} + \frac{2}{r}\frac{\partial f}{\partial r} = \frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial f}{\partial r}\right) \tag{6.33}$$

and

$$\frac{\partial^2 f}{\partial\theta^2} + \frac{\cos\theta}{\sin\theta}\frac{\partial f}{\partial\theta} = \frac{1}{\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial f}{\partial\theta}\right) \tag{6.34}$$

We need to avoid $r = 0$ and $\sin\theta = 0$. The latter condition means avoiding $\theta = 0$ and $\theta = \pi$.

Gradient of a Scalar Field

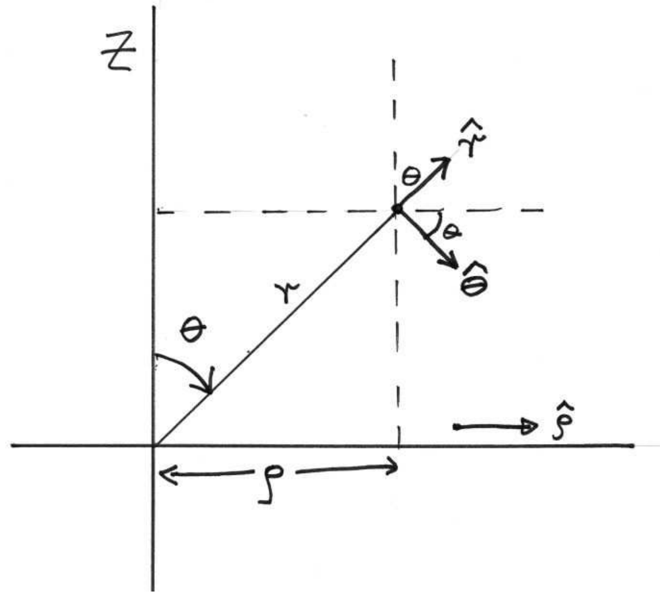

Figure 20: $\mathbf{r}$ and $\boldsymbol{\theta}$ unit vectors

The batch file sphere.mac introduces the cartesian and spherical polar unit vectors corresponding to the coordinates (r, θ, φ). The unit vector $\hat{\boldsymbol{\varphi}}$ is the same as in cylindrical coordinates:

$$\hat{\boldsymbol{\varphi}} = -\hat{\mathbf{x}}\,\sin\varphi + \hat{\mathbf{y}}\,\cos\varphi \tag{6.35}$$

We can express $\hat{\mathbf{r}}$ and $\hat{\boldsymbol{\theta}}$ in terms of the cylindrical $\hat{\boldsymbol{\rho}}$ and $\hat{\mathbf{z}}$ (from the figure above):

$$\hat{\mathbf{r}} = \hat{\mathbf{z}}\,\cos\theta + \hat{\boldsymbol{\rho}}\,\sin\theta \tag{6.36}$$

and

$$\hat{\boldsymbol{\theta}} = -\hat{\mathbf{z}}\,\sin\theta + \hat{\boldsymbol{\rho}}\,\cos\theta. \tag{6.37}$$

Hence we have

$$\hat{\mathbf{r}} = \hat{\mathbf{x}}\,\sin\theta\,\cos\varphi + \hat{\mathbf{y}}\,\sin\theta\sin\varphi + \hat{\mathbf{z}}\,\cos\theta \tag{6.38}$$

and

$$\hat{\boldsymbol{\theta}} = \hat{\mathbf{x}}\,\cos\theta\cos\varphi + \hat{\mathbf{y}}\,\cos\theta\,\sin\varphi - \hat{\mathbf{z}}\,\sin\theta \tag{6.39}$$

Maxima has already been told to treat f as an explicit function of **(r,t,p)** which we are using for (r, θ, φ). The vector components of $\boldsymbol{\nabla} f$ are isolated by using the dot product of the unit vectors with $\boldsymbol{\nabla} f$. For example,

$$(\boldsymbol{\nabla} f)_r = \hat{\mathbf{r}} \cdot \boldsymbol{\nabla} f \tag{6.40}$$

```
(%i35)          --------------------------------------------
(%i36)                      Unit Vectors
(%i37)          --------------------------------------------
(%i38)                   cartesian unit vectors
(%i39)        (xu : [1, 0, 0], yu : [0, 1, 0], zu : [0, 0, 1])
(%i40)           spherical polar coordinate unit vectors
(%i41)      ru : cos(t) zu + sin(t) sin(p) yu + sin(t) cos(p) xu
(%o41)            [cos(p) sin(t), sin(p) sin(t), cos(t)]
(%i42)     tu : - sin(t) zu + cos(t) sin(p) yu + cos(t) cos(p) xu
(%o42)           [cos(p) cos(t), sin(p) cos(t), - sin(t)]
(%i43)                 pu : cos(p) yu - sin(p) xu
(%o43)                    [- sin(p), cos(p), 0]
```

```
(%i44)          -------------------------------------------
(%i45)                  Gradient of a Scalar Function f
(%i46)          -------------------------------------------
(%i47)             cartesian def. of gradient of scalar f
(%i48)    fgradient : diff(f, z) zu + diff(f, y) yu + diff(f, x) xu
(%i49)             r, theta, and phi components of grad(f)
(%i50)          fgradient_r : (ru . fgradient, trigsimp(%%))
                                     df
(%o50)                               --
                                     dr
(%i51)          fgradient_t : (tu . fgradient, trigsimp(%%))
                                     df
                                     --
                                     dt
(%o51)                               --
                                     r
(%i52)          fgradient_p : (pu . fgradient, trigsimp(%%))
                                     df
                                     --
                                     dp
(%o52)                             --------
                                   r sin(t)
```

Thus we have the gradient of a scalar field in spherical polar coordinates:

$$\boldsymbol{\nabla} f(r, \theta, \varphi) = \hat{\mathbf{r}} \frac{\partial f}{\partial r} + \hat{\boldsymbol{\theta}} \frac{1}{r} \frac{\partial f}{\partial \theta} + \hat{\boldsymbol{\varphi}} \frac{1}{r \sin\theta} \frac{\partial f}{\partial \varphi} \tag{6.41}$$

Divergence of a Vector Field

The path here is the same as in the cylindrical case. For example, we define the spherical polar components of the vector $\mathbf{B}$ via

$$B_r = \hat{\mathbf{r}} \cdot \mathbf{B}, B_\theta = \hat{\boldsymbol{\theta}} \cdot \mathbf{B}, B_\varphi = \hat{\boldsymbol{\varphi}} \cdot \mathbf{B}. \tag{6.42}$$

```
(%i53)         -----------------------------------------------
(%i54)               Divergence of a Vector bvec
(%i55)         -----------------------------------------------
(%i56)                  bvec : bz zu + by yu + bx xu
(%o56)                          [bx, by, bz]
(%i57)      three equations which relate spherical polar components
(%i58)                of bvec to the cartesian components
(%i59)                      eq1 : br = ru . bvec
(%o59)       br = by sin(p) sin(t) + bx cos(p) sin(t) + bz cos(t)
(%i60)                      eq2 : bt = tu . bvec
(%o60)      bt = - bz sin(t) + by sin(p) cos(t) + bx cos(p) cos(t)
(%i61)                      eq3 : bp = pu . bvec
(%o61)                   bp = by cos(p) - bx sin(p)
(%i62)                     invert these equations
(%i63)   sol : (linsolve([eq1, eq2, eq3], [bx, by, bz]), trigsimp(%%))
(%o63) [bx = br cos(p) sin(t) + bt cos(p) cos(t) - bp sin(p),
by = br sin(p) sin(t) + bt sin(p) cos(t) + bp cos(p),
bz = br cos(t) - bt sin(t)]
(%i64)                  [bx, by, bz] : map('rhs, sol)
(%i65)        tell Maxima to treat spherical polar components as
(%i66)                   explicit functions of (r,t,p)
(%i67)                 depends([br, bt, bp], [r, t, p])
```

```
(%i68)                divergence of bvec
(%i69) bdivergence : (diff(bz, z) + diff(by, y) + diff(bx, x), trigsimp(%%),
                                                         scanmap('multthru, %%))
                             dbp      dbt
                             ---      ---
                  bt cos(t)  dp       dt     2 br   dbr
(%o69)            --------- + -------- + --- + ---- + ---
                  r sin(t)   r sin(t)  r      r     dr
```

Hence we have the spherical polar coordinate version of the divergence of a vector field:

$$\nabla \cdot \mathbf{B}(r,\theta,\varphi) = \frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\,B_r\right) + \frac{1}{r\,\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\,B_\theta\right) + \frac{1}{r\,\sin\theta}\frac{\partial B_\varphi}{\partial\varphi} \tag{6.43}$$

using the identities

$$\frac{2}{r}\,B_r + \frac{\partial B_r}{\partial r} = \frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\,B_r\right) \tag{6.44}$$

and

$$\frac{\cos\theta}{\sin\theta}\,B_\theta + \frac{\partial B_\theta}{\partial\theta} = \frac{1}{\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\,B_\theta\right) \tag{6.45}$$

Curl of a Vector Field

Again the path is essentially the same as in the cylindrical case:

```
(%i70)     --------------------------------------------------------
(%i71)        Spherical Polar Components of Curl(bvec)
(%i72)     --------------------------------------------------------
(%i73)                  cartesian curl(bvec) definition
(%i74) bcurl : (diff(by, x) - diff(bx, y)) zu + (diff(bx, z) - diff(bz, x)) yu
                                             + (diff(bz, y) - diff(by, z)) xu
(%i75)                spherical polar components of curl(bvec)
(%i76)   bcurl_r : (ru . bcurl, trigsimp(%%), scanmap('multthru, %%))
                                        dbt      dbp
                                        ---      ---
                        bp cos(t)       dp       dt
(%o76)                  --------- - -------- + ---
                        r sin(t)    r sin(t)    r
(%i77)   bcurl_t : (tu . bcurl, trigsimp(%%), scanmap('multthru, %%))
                                dbr
                                ---
                                dp       bp   dbp
(%o77)                        -------- - -- - ---
                              r sin(t)   r    dr
(%i78)   bcurl_p : (pu . bcurl, trigsimp(%%), scanmap('multthru, %%))
                                     dbr
                                     ---
                                bt   dt    dbt
(%o78)                          -- - --- + ---
                                r    r     dr
(%i79)      -------------------------------------------------------
```

which produces the spherical polar coordinate version of the curl of a vector field:

$$(\nabla\times\mathbf{B})_r = \frac{1}{r\,\sin\theta}\left[\frac{\partial}{\partial\theta}\left(\sin\theta\,B_\varphi\right) - \frac{\partial B_\theta}{\partial\varphi}\right] \tag{6.46}$$

$$(\nabla\times\mathbf{B})_\theta = \frac{1}{r}\left[\frac{1}{\sin\theta}\frac{\partial B_r}{\partial\varphi} - \frac{\partial}{\partial r}\left(r\,B_\varphi\right)\right] \tag{6.47}$$

$$(\nabla\times\mathbf{B})_\varphi = \frac{1}{r}\left[\frac{\partial}{\partial r}\left(r\,B_\theta\right) - \frac{\partial B_r}{\partial\theta}\right] \tag{6.48}$$

in which we have used Eq.(6.45) with B_θ replaced by B_φ and also Eq.(6.31) with ρ replaced by r.

Maxima by Example:
Ch.7: Symbolic Integration *

Edwin L. Woollett

September 16, 2010

Contents

7.1 Symbolic Integration with **integrate** 3
7.2 Integration Examples and also **defint**, **ldefint**, **beta**, **gamma**, **erf**, and **logabs** 4
7.3 Piecewise Defined Functions and **integrate** 12
7.4 Area Between Curves Examples 14
7.5 Arc Length of an Ellipse 17
7.6 Double Integrals and the Area of an Ellipse 19
7.7 Triple Integrals: Volume and Moment of Inertia of a Solid Ellipsoid 22
7.8 Derivative of a Definite Integral with Respect to a Parameter 24
7.9 Integration by Parts 28
7.10 Change of Variable and **changevar** 29

*This version uses **Maxima 5.18.1**. This is a live document. Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions to `woollett@charter.net`

```
COPYING AND DISTRIBUTION POLICY
This document is part of a series of notes titled
"Maxima by Example" and is made available
via the author's webpage http://www.csulb.edu/~woollett/
to aid new users of the Maxima computer algebra system.

NON-PROFIT PRINTING AND DISTRIBUTION IS PERMITTED.

You may make copies of this document and distribute them
to others as long as you charge no more than the costs of printing.

These notes (with some modifications) will be published in book form
eventually via Lulu.com in an arrangement which will continue
to allow unlimited free download of the pdf files as well as the option
of ordering a low cost paperbound version of these notes.
```


Feedback from readers is the best way for this series of notes to become more helpful to new users of Maxima. *All* comments and suggestions for improvements will be appreciated and carefully considered.

```
The Maxima session transcripts were generated using the XMaxima
graphics interface on a Windows XP computer, and copied into
a fancy verbatim environment in a latex file which uses the
fancyvrb and color packages.

We use qdraw.mac (available on the author's web page with Ch. 5 materials)
for plots.
```


```
Maxima.sourceforge.net. Maxima, a Computer Algebra System. Version 5.18.1
 (2009). http://maxima.sourceforge.net/
```

7.1 Symbolic Integration with integrate

Although the process of finding an integral can be viewed as the inverse of the process of finding a derivative, in practice finding an integral is more difficult. It turns out that the integral of fairly simple looking functions cannot be expressed in terms of well known functions, and this has been one motivation for the introduction of definitions for a variety of "special functions", as well as efficient methods for numerical integration (quadrature) discussed in Ch. 8.

In the following, we always assume we are dealing with a real function of a real variable and that the function is single-valued and continuous over the intervals of interest.

The Maxima manual entry for **integrate** includes (we have made some changes and additions):

> Function: **integrate(expr, var)**
> Function: **integrate(expr, var, a, b)**
> Attempts to symbolically compute the integral of **expr** with respect to **var**. **integrate(expr, var)** is an indefinite integral, while **integrate(expr, var, a, b)** is a definite integral, with limits of integration **a** and **b**. The integral is returned if **integrate** succeeds. Otherwise the return value is the noun form of the integral (the quoted operator **'integrate**) or an expression containing one or more noun forms. The noun form of **integrate** is displayed with an integral sign if **display2d** is set to **true** (which is the default).
>
> In some circumstances it is useful to construct a noun form by hand, by quoting **integrate** with a single quote, e.g., **'integrate(expr, var)**. For example, the integral may depend on some parameters which are not yet computed. The noun may be applied to its arguments by **ev(iexp, nouns)** where **iexp** is the noun form of interest.
>
> The Maxima function **integrate** is defined by the lisp function **$integrate** in the file /src/simp.lisp. The indefinite integral invocation, **integrate(expr,var)**, results in a call to the lisp function **sinint**, defined in src/sin.lisp, unless the flag **risch** is present, in which case the lisp function **rischint**, defined in src/risch.lisp, is called. The definite integral invocation, **integrate(expr,var,a,b)**, causes a call to the lisp function **$defint**, defined in src/defint.lisp. The lisp function **$defint** is available as the Maxima function **defint** and can be used to bypass **integrate** for a definite integral.
>
> To integrate a Maxima function $\mathbf{f(x)}$, insert $\mathbf{f(x)}$ in the **expr** slot.
>
> **integrate** does not respect implicit dependencies established by the **depends** function.
>
> **integrate** may need to know some property of the parameters in the integrand. **integrate** will first consult the **assume** database, and, if the variable of interest is not there, **integrate** will ask the user. Depending on the question, suitable responses are **yes;** or **no;**, or **pos;**, **zero;**, or **neg;**. Thus, the user can use the **assume** function to avoid all or some questions.

7.2 Integration Examples and also defint, ldefint, beta, gamma, erf, and logabs

Example 1

Our first example is the indefinite integral $\int \sin^3 x \, dx$:

```
(%i1) integrate (sin(x)^3, x);
                                  3
                               cos  (x)
(%o1)                          ------- - cos(x)
                                  3
(%i2) ( diff (%,x), trigsimp(%%) );
                                       3
(%o2)                               sin  (x)
```

Notice that the indefinite integral returned by **integrate** *does not include* the arbitrary constant of integration which can always be added.

If the returned integral is correct (up to an arbitrary constant), then the first derivative of the returned indefinite integral should be the original integrand, although we may have to simplify the result manually (as we had to do above).

Example 2

Our second example is another indefinite integral, $\int x \, (b^2 - x^2)^{-1/2} \, dx$:

```
(%i3) integrate (x/ sqrt (b^2 - x^2), x);
                                        2    2
(%o3)                           - sqrt(b  - x )
(%i4) diff(%,x);
                                       x
(%o4)                            -------------
                                       2    2
                                 sqrt(b  - x )
```

Example 3

The definite integral can be related to the "area under a curve" and is the more accessible concept, while the integral is simply a function whose first derivative is the original integrand.

Here is a definite integral, $\int_0^{\pi} \cos^2 x \, e^x \, dx$:

```
(%i5) i3 : integrate (cos(x)^2 * exp(x), x, 0, %pi);

                                     %pi
                                 3 %e       3
(%o5)                            ------- - -
                                    5       5
```

Instead of using **integrate** for a definite integral, you can try **ldefint** (think Limit definite integral), which may provide an alternative form of the answer (if successful).

From the Maxima manual:

Function: **ldefint(expr, x, a, b)**
Attempts to compute the definite integral of **expr** by using **limit** to evaluate the indefinite integral of **expr** with respect to x at the upper limit b and at the lower limit a. If it fails to compute the definite integral, **ldefint** returns an expression containing limits as noun forms.

ldefint is not called from **integrate**, so executing **ldefint(expr, x, a, b)** may yield a different result than **integrate(expr, x, a, b)**. **ldefint** always uses the same method to evaluate the definite integral, while **integrate** may employ various heuristics and may recognize some special cases.

Here is an example of use of **ldefint**, as well as the direct use of **defint** (which bypasses **integrate**):

```
(%i6) ldefint (cos(x)^2 * exp(x), x, 0, %pi);
                                        %pi
                                    3 %e       3
(%o6)                               ------- - -
                                       5       5
(%i7) defint (cos(x)^2 * exp(x), x, 0, %pi);
                                        %pi
                                    3 %e       3
(%o7)                               ------- - -
                                       5       5
```

Example 4

Here is an example of a definite integral over an infinite range, $\int_{-\infty}^{\infty} x^2\, e^{-x^2}\, dx$:

```
(%i8) integrate (x^2 * exp(-x^2), x, minf, inf);
                                     sqrt(%pi)
(%o8)                                ---------
                                         2
```

To check this integral, we first ask for the indefinite integral and then check it by differentiation.

```
(%i9) i1 : integrate(x^2*exp(-x^2),x );
                                                        2
                                                     - x
                              sqrt(%pi) erf(x)    x %e
(%o9)                         ---------------- - --------
                                     4               2
(%i10) diff(i1,x);
                                              2
                                      2    - x
(%o10)                               x   %e
```

Thus the indefinite integral is correct. The second term, heavily damped by the factor e^{-x^2} at $\pm\infty$, does not contribute to the definite integral. The first term is proportional to the (Gauss) error function, $\mathrm{erf}(x)$, in which x is real. For (in general) complex $w = u + i\, v$,

$$\mathrm{Erf}(w) = \frac{2}{\sqrt{\pi}} \int_0^w e^{-z^2}\, dz \tag{7.1}$$

in which we can integrate over any path connecting 0 and w in the complex $z = x + i\, y$ plane, since the integrand is an entire function of z (no singularities in the finite z plane.

Let's make a plot of **erf(x)**:

```
(%i11) ( load(draw),load(qdraw) )$
(%i12) qdraw(yr(-2,2),ex1(erf(x),x,-5,5,lw(5),lc(red),lk("erf(x)") ) )$
```

with the result:

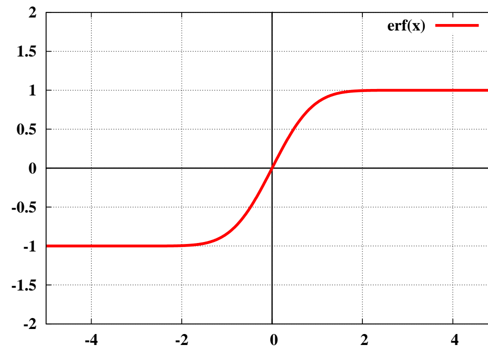

Figure 1: $\mathrm{erf}(x)$

Maxima's **limit** function confirms what the plot indicates:

```
(%i13) [limit(erf(x),x, inf), limit(erf(x),x, minf)];
(%o13)                             [1, - 1]
```

Using these limits in **%o9** produces the definite integral desired.

Example 5: Use of assume

We next calculate the definite integral $\int_0^\infty x^a\,(x+1)^{-5/2}\,dx$.

```
(%i1) (assume(a>1),facts());
(%o1)                               [a > 1]
(%i2) integrate (x^a/(x+1)^(5/2), x, 0, inf );
   2 a + 2
Is ------- an integer?
      5

no;
Is  2 a - 3  positive, negative, or zero?

neg;
                                      3
(%o2)                          beta(- - a, a + 1)
                                      2
```

The combination of $\mathrm{assume}(a > 1)$ and $2\,a - 3 < 0$ means that we are assuming $1 < a < 3/2$. These assumptions about a imply that $4/5 < (2\,a + 2)/5 < 1$. To be consistent, we must hence answer **no** to the first question.

Let's tell Maxima to **forget** about the **assume** assignment and see what the difference is.

```
(%i3) ( forget(a>1), facts() );
(%o3)                                 []
(%i4) is( a>1 );
(%o4)                               unknown
(%i5) integrate (x^a/(x+1)^(5/2), x, 0, inf );
Is  a + 1  positive, negative, or zero?

pos;
Is a an integer?

no;
      7
Is ------- an integer?
   2 a + 4

no;
Is  2 a - 3  positive, negative, or zero?

neg;
                                   3
(%o5)                         beta(- - a, a + 1)
                                   2
(%i6) [is( a>1 ), facts() ];
(%o6)                            [unknown, []]
```

Thus we see that omitting the initial `assume(a>1)` statement causes **integrate** to ask four questions instead of two. We also see that answering questions posed by the **integrate** dialogue script does not result in population of the **facts** list.

The Maxima **beta** function has the manual entry:

```
Function: beta (a, b)
The beta function is defined as gamma(a) gamma(b)/gamma(a+b) (A&S 6.2.1).
```

In the usual mathematics notation, the beta function can be defined in terms of the gamma function as

$$B(r, s) = \frac{\Gamma(r)\,\Gamma(s)}{\Gamma(r+s)} \tag{7.2}$$

for all r, s in the complex plane.

The Maxima **gamma** function has the manual entry

```
Function: gamma (z)
The basic definition of the gamma function (A&S 6.1.1) is

                                 inf
                                /
                                [     z - 1   - t
                   gamma(z) =   I    t       %e    dt
                                ]
                                /
                                 0
```

The gamma function can be defined for complex z and $\mathrm{Re}(z) > 0$ by the integral along the real t axis

$$\Gamma(z) = \int_0^{\infty} t^{z-1}\, e^{-t}\, dt \tag{7.3}$$

and for $\mathrm{Im}(z) = 0$ and $\mathrm{Re}(z) = n$ and n an integer greater than zero we have

$$\Gamma(n+1) = n! \tag{7.4}$$

How can we check the definite integral Maxima has offered? If we ask the **integrate** function for the indefinite integral, we get the "noun form", a signal of failure:

```
(%i7) integrate(x^a/(x+1)^(5/2), x );
                               /      a
                               [     x
(%o7)                          I ---------- dx
                               ]        5/2
                               /  (x + 1)
(%i8) grind(%)$
'integrate(x^a/(x+1)^(5/2),x)$
```

Just for fun, let's include the **risch** flag and see what we get:

```
(%i9) integrate(x^a/(x+1)^(5/2), x ), risch;
                       /           a log(x)
                       [         %e
(%o9)                  I -------------------------- dx
                       ]                2
                       / sqrt(x + 1) (x  + 2 x + 1)
```

We again are presented with a noun form, but the integrand has been written in a different form, in which the identity

$$x^A = e^{A \ln(x)}$$

has been used.

We can at least make a spot check for a value of the parameter a in the middle of the assumed range $(1, 3/2)$, namely for $a = 5/4$.

```
(%i10) float(beta(1/4,9/4));
(%o10)                          3.090124462168955
(%i11) quad_qagi(x^(5/4)/(x+1)^(5/2),x, 0, inf);
(%o11)        [3.090124462010259, 8.6105700347616221E-10, 435, 0]
```

We have used the quadrature routine, **quad_qagi** (see Ch. 8) for a numerial estimate of this integral. The first element of the returned list is the numerical answer, which agrees with the analytic answer.

Example 6: Automatic Change of Variable and gradef

Here is an example which illustrates Maxima's ability to make a change of variable to enable the return of an indefinite integral. The task is to evaluate the indefinite integral

$$\int \frac{\sin(r^2)\, dr(x)/dx}{q(r)}\, dx \tag{7.5}$$

by telling Maxima that the $\sin(r^2)$ in the numerator is related to $q(r)$ in the denominator by the derivative:

$$\frac{d\, q(u)}{d\, u} = \sin(u^2). \tag{7.6}$$

We would manually rewrite the integrand (using the chain rule) as

$$\frac{\sin(r^2)\, dr(x)/dx}{q} = \frac{1}{q}\, (dq(r)/dr)\, (dr(x)/dx) = \frac{1}{q}\frac{dq}{dx} = \frac{d}{dx}\ln(q) \tag{7.7}$$

and hence obtain the indefinite integral $\ln(q(r(x)))$.

Here we assign the derivative knowledge using **gradef** (as discussed in Ch. 6):

```
(%i1) gradef(q(u),sin(u^2) )$
(%i2) integrand : sin(r(x)^2)* 'diff(r(x),x ) /q( r(x) ) ;
                               d                2
                              (-- (r(x))) sin(r  (x))
                               dx
(%o2)                         -----------------------
                                    q(r(x))
(%i3) integrate(integrand,x);
(%o3)                             log(q(r(x)))
(%i4) diff(%,x);
                               d                2
                              (-- (r(x))) sin(r  (x))
                               dx
(%o4)                         -----------------------
                                    q(r(x))
```

Note that **integrate** pays no attention to **depend** assignments, so the briefer type of notation which **depend** allows with differentiation cannot be used with **integrate**:

```
(%i5) depends(r,x,q,r);
(%o5)                             [r(x), q(r)]
(%i6) integrand : sin(r^2)* 'diff(r,x) / q;
                                  dr      2
                                  -- sin(r )
                                  dx
(%o6)                             ----------
                                      q
(%i7) integrate(integrand,x);
                                        /
                                     2  [ dr
                                 sin(r ) I -- dx
                                        ] dx
                                        /
(%o7)                            ---------------
                                       q
```

which is fatally flawed, since Maxima pulled both $\sin(r(x)^2)$ and $1/q(r(x))$ outside the integral.

Of course, the above **depends** assignment will still allow Maxima to rewrite the derivative of q with respect to x using the chain rule:

```
(%i8) diff(q,x);
                                     dq dr
(%o8)                                -- --
                                     dr dx
```

Example 7: Integration of Rational Algebraic Functions, rat, ratsimp, and partfrac

A *rational algebraic function* can be written as a quotient of two polynomials. Consider the following function of x.

```
(%i1) e1 : x^2 + 3*x -2/(3*x)+110/(3*(x-3)) + 12;
                             2           2       110
(%o1)                       x  + 3 x - --- + --------- + 12
                                       3 x   3 (x - 3)
```

We can obviously find the lowest common denominator and write this as the ratio of two polynomials, using either **rat** or **ratsimp**.

```
(%i2) e11 : ratsimp(e1);
                                      4      2
                                     x  + 3 x  + 2
(%o2)                                -------------
                                        2
                                       x  - 3 x
```

Because the polynomial in the numerator is of higher degree than the polynomial in the denominator, this is called an *improper rational fraction*. Any improper rational fraction can be reduced by division to a mixed form, consisting of a sum of some polynomial and a sum of proper fractions. We can recover the "partial fraction" representation in terms of *proper rational fractions* (numerator degree less than denominator degree) by using **partfrac** `(expr, var)`.

```
(%i3) e12 : partfrac(e11,x);
                             2           2       110
(%o3)                       x  + 3 x - --- + --------- + 12
                                       3 x   3 (x - 3)
```

With this function of x expressed in partial fraction form, you are able to write down the indefinite integral immediately (ie., by inspection, without using Maxima). But, of course, we want to practice using Maxima!

```
(%i4) integrate(e11,x);
                                                        3      2
                2 log(x)    110 log(x - 3)    2 x  + 9 x  + 72 x
(%o4)         - -------- + -------------- + ------------------
                   3             3                   6
(%i5) integrate(e12,x);
                                                      3      2
                 2 log(x)    110 log(x - 3)    x     3 x
(%o5)          - -------- + -------------- + -- + ---- + 12 x
                    3             3           3     2
```

Maxima has to do less work if you have already provided the partial fraction form as the integrand; otherwise, Maxima internally seeks a partial fraction form in order to do the integral.

Example 8

The next example shows that **integrate** can sometimes split the integrand into at least one piece which can be integrated, and leaves the remainder as a formal expression (using the noun form of **integrate**). This may be possible if the denominator of the integrand is a polynomial which Maxima can factor.

```
(%i6) e2: 1/(x^4 - 4*x^3 + 2*x^2 - 7*x - 4);
                                        1
(%o6)                     --------------------------
                            4      3      2
                           x  - 4 x  + 2 x  - 7 x - 4
(%i7) integrate(e2,x);
                                       /  2
                                       [ x  + 4 x + 18
                                       I ------------- dx
                                       ]  3
                          log(x - 4)   / x  + 2 x + 1
(%o7)                    ---------- - ------------------
                              73                73
(%i8) grind(%)$
log(x-4)/73-('integrate((x^2+4*x+18)/(x^3+2*x+1),x))/73$
(%i9) factor(e2);
                                        1
(%o9)                       ----------------------
                                        3
                             (x - 4) (x  + 2 x + 1)
(%i10) partfrac(e2,x);
                                          2
                             1           x  + 4 x + 18
(%o10)                    ---------- - ----------------
                          73 (x - 4)          3
                                       73 (x  + 2 x + 1)
```

We have first seen what Maxima can do with this integrand, using the **grind** function to clarify the resulting expression, and then we have used **factor** and **partfrac** to see how the split-up arises. Despite a theorem that the integral of every rational function can be expressed in terms of algebraic, logarithmic and inverse trigonometric expressions, Maxima declines to return a symbolic expression for the second, formal, piece of this integral (which is good because the exact symbolic answer is an extremely long expression).

Example 9: The logabs Parameter and log

There is a global parameter **logabs** whose default value is **false** and which affects what is returned with an indefinite integral containing logs.

```
(%i11) logabs;
(%o11)                                  false
(%i12) integrate(1/x,x);
(%o12)                                 log(x)
(%i13) diff(%,x);
                                          1
(%o13)                                    -
                                          x
(%i14) logabs:true$
```

```
(%i15) integrate(1/x,x);
(%o15)                           log(abs(x))
(%i16) diff(%,x);
                                      1
(%o16)                                -
                                      x
(%i17) log(-1);
(%o17)                            log(- 1)
(%i18) float(%);
(%o18)                      3.141592653589793 %i
```

When we override the default and set **logabs** to **true**, the argument of the **log** function is wrapped with **abs**. According to the manual

> When doing indefinite integration where logs are generated, e.g. integrate(1/x,x), the answer is given in terms of log(abs(...)) if logabs is true, but in terms of log(...) if logabs is false. For definite integration, the logabs:true setting is used, because here "evaluation" of the indefinite integral at the endpoints is often needed.

7.3 Piecewise Defined Functions and integrate

We can use Maxima's **if**, **elseif**, and **else** construct to define piecewise defined functions. We first define and plot a square wave of unit height which extends from $1 \le x \le 3$.

```
(%i1) u(x) := if x >= 1 and x <= 3 then 1 else 0$
(%i2) map('u,[0.5,1,2,3,3.5]);
(%o2)                             [0, 1, 1, 1, 0]
(%i3) (load(draw),load(qdraw))$
               qdraw(...), qdensity(...), syntax: type qdraw();
(%i4) qdraw( yr(-1,2),ex1(u(x),x,0,4,lw(5),lc(blue)) )$
```

This produces

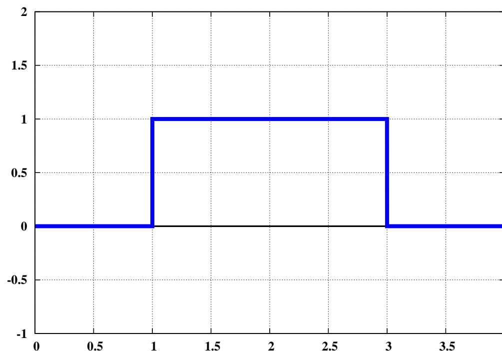

Figure 2: if $x > 1$ and $x < 3$ then 1 else 0

We next define a function which is $(x - 1)$ for $1 \leq x < 2$ and is $(6 - 2\,x)$ for $2 \leq x \leq 3$ and is 0 otherwise.

```
(%i5) g(x):= if x >= 1 and x < 2 then (x-1)
             elseif x >= 2 and x <= 3 then (6 - 2*x) else 0$
(%i6) map('g,[1/2,1,3/2,2,5/2,3,7/2]);
                                     1
(%o6)                         [0, 0, -, 2, 1, 0, 0]
                                     2
(%i7) qdraw( yr(-1,3),ex1(g(x),x,0,4,lw(5),lc(blue)) )$
```

which produces

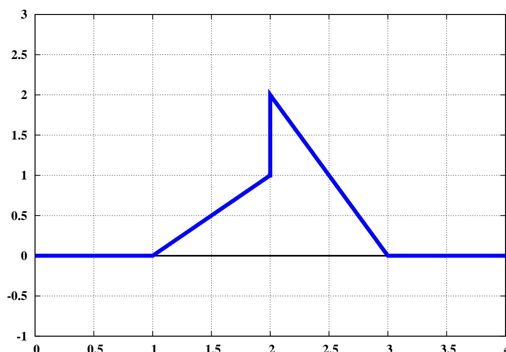

Figure 3: Example of a Piecewise Defined Function

This is not quite the figure we want, since it doesn't really show the kind of discontinuity we want to illustrate. With our definition of $g(x)$, **draw2d** draws a blue vertical line from $(2, 1)$ to $(2, 2)$. We can change the way $g(x)$ is plotted to get the plot we want, as in

```
(%i8) block([small :1.0e-6],
         qdraw( yr(-1,3),ex1(g(x),x,0,2-small,lw(5),lc(blue)),
                  ex1(g(x),x,2+small,4,lw(5),lc(blue)) ) )$
```

which produces

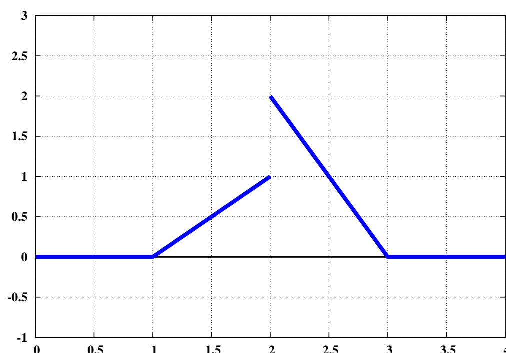

Figure 4: Example with More Care with Plot Limits

The Maxima function **integrate** cannot handle correctly a function defined with the **if**, **elseif**, and **else** constructs.

```
(%i9) integrate(g(x),x,0,4);
        4
       /
       [
(%o9) I   (if (x > 1) and (x < 2) then x - 1 elseif (x > 2) and (x < 3)
       ]
       /
        0
                                                   then 6 - 2 x else 0) dx
```

This means that for now we will have to break up the integration interval into sub-intervals by hand and add up the individual integral results.

```
(%i10) integrate(x-1,x,1,2) + integrate(6-2*x,x,2,3);
                                        3
(%o10)                                  -
                                        2
```

7.4 Area Between Curves Examples

Example 1

We will start with a very simple example, finding the area between $f_1(x) = \sqrt{x}$ and $f_2(x) = x^{3/2}$. A simple plot shows that the curves cross at $x = 0$ and $x = 1$.

```
(%i1) (load(draw),load(qdraw) )$
(%i2) f1(x) := sqrt(x)$
(%i3) f2(x) := x^(3/2)$
(%i4) qdraw(   xr(-.5,1.5), yr(-.5,1.5),
         ex1(f1(x),x,0,1.5,lw(5),lc(blue) ),
         ex1(f2(x),x,0,1.5,lw(5),lc(red) ) )$
```

This produces the plot:

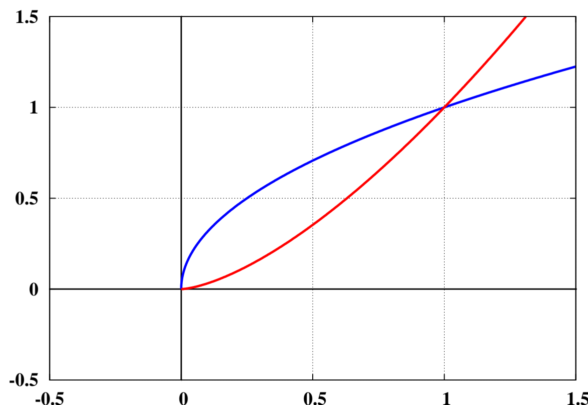

Figure 5: $\sqrt{x}$ (blue) and $x^{3/2}$ (red)

Next we redraw this simple plot, but add some shading in color to show the area. The simplest way to do this is to draw some vertical lines between the functions in the region of interest $0 \le x \le 1$. We can use the **qdraw** package function **line**. We first construct a list of x axis positions for the vertical lines, using values $0.01, 0.02, ...0.99$. We then construct a list `vv` of the vertical lines and merge that list with a list `qdlist` containing the rest of the plot instructions. We then use **apply** to pass this list as a set of arguments to **qdraw**.

```
(%i5) qdlist : [xr(-.5,1.5), yr(-.5,1.5),
          ex1(f1(x),x,0,1.5,lw(5),lc(blue) ) ,
          ex1(f2(x),x,0,1.5,lw(5),lc(red) ) ]$
(%i6) xv:float(makelist(i,i,1,99)/100)$
(%i7) (vv:[],for x in xv do
        vv:cons(line(x,f2(x),x,f1(x),lw(1),lc(khaki) ),vv),
        vv:reverse(vv) )$
(%i8) qdlist : append(vv,qdlist)$
(%i9) apply('qdraw, qdlist)$
```

which produces the result

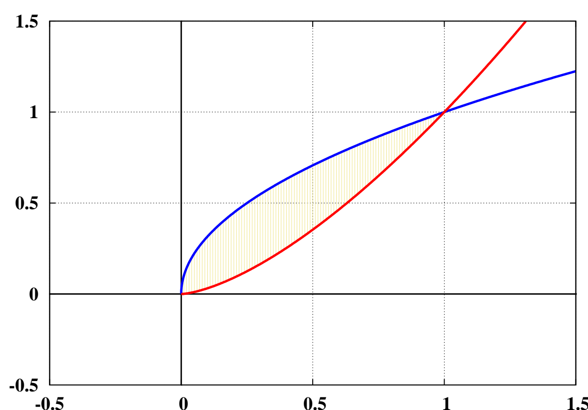

Figure 6: $\sqrt{x}$ (blue) and $x^{3/2}$ (red)

If we did not know the intersection location of the two curves, we could use **solve** or **`find_root`** for example.

```
(%i10) solve( f1(x) = f2(x),x );
(%o10)                         [x = 0, x = 1]
```

Once we know the interval to use for adding up the area and we know that in this interval $f_1(x) > f_2(x)$, we simply sum the infinitesimal areas given by $(f_1(x) - f_2(x))\,dx$ (base dx columns) over the interval $0 \le x \le 1$.

```
(%i11) integrate(f1(x) - f2(x),x,0,1);
                                    4
(%o11)                             ----
                                    15
```

so the area equals $4/15$.

$$\int_0^1 (\sqrt{x} - x^{3/2})\,dx = 4/15 \tag{7.8}$$

Example 2

As a second example we consider two polynomial functions:
$f_1(x) = (3/10)\,x^5 - 3\,x^4 + 11\,x^3 - 18\,x^2 + 12\,x + 1$
and $f_2(x) = -4\,x^3 + 28\,x^2 - 56\,x + 32$. We first make a simple plot for orientation.

```
(%i1) f1(x) := (3/10)*x^5 -3*x^4 + 11*x^3 -18*x^2 + 12*x + 1$
(%i2) f2(x) := -4*x^3 + 28*x^2 -56*x + 32$
(%i3) (load(draw),load(qdraw) )$
                  qdraw(...), qdensity(...), syntax: type qdraw();

(%i4) qdraw(yr(-20,20),ex1(f1(x),x,-1,5,lc(blue) ),
                  ex1(f2(x),x,-1,5,lc(red)))$
```

which produces the plot

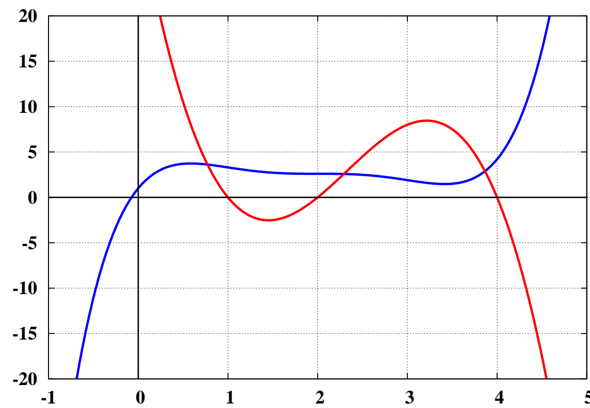

Figure 7: $f_1(x)$ (blue) and $f_2(x)$ (red)

Using the cursor on the plot, and working from left to right, f_1 becomes larger than f_2 at about $(0.76, 3.6)$, becomes less than f_2 at about $(2.3, 2.62)$, and becomes greater than f_2 at about $(3.86, 2.98)$. The **solve** function is not able to produce an analytic solution, but returns a polynomial whose roots are the solutions we want.

```
(%i5) solve(f1(x) = f2(x),x);
                             5        4         3         2
(%o5)          [0 = 3 x  - 30 x  + 150 x  - 460 x  + 680 x - 310]
(%i6) grind(%)$
[0 = 3*x^5-30*x^4+150*x^3-460*x^2+680*x-310]$
```

By selecting and copying the **grind(..)** output, we can use that result to paste in the definition of a function $p(x)$ which we can then use with `find_root`.

```
(%i7) p(x) := 3*x^5-30*x^4+150*x^3-460*x^2+680*x-310$
(%i8) x1 : find_root(p(x),x,.7,1);
(%o8)                         0.77205830452781
(%i9) x2 : find_root(p(x),x,2.2,2.4);
(%o9)                         2.291819210962957
(%i10) x3 : find_root(p(x),x,3.7,3.9);
(%o10)                         3.865127100061791
(%i11) map('p, [x1,x2,x3] );
(%o11)                 [0.0, 0.0, 9.094947017729282E-13]
(%i12) [y1,y2,y3] : map('f1, [x1,x2,x3] );
(%o12)      [3.613992056691179, 2.575784006305792, 2.882949345140702]
```

We have checked the solutions by seeing how close to zero $p(x)$ is when x is one of the three roots $[x1, x2, x3]$. We now split up the integration into the two separate regions where one or the other function is larger.

```
(%i13) ratprint:false$
(%i14) i1 : integrate(f1(x) - f2(x),x,x1,x2);
                                    41875933
(%o14)                              --------
                                    7947418
(%i15) i2 : integrate(f2(x)-f1(x),x,x2,x3);
                                    12061231
(%o15)                              --------
                                    1741444
(%i16) area : i1 + i2;
                                  30432786985
(%o16)                            -----------
                                  2495489252
(%i17) area : float(area);
(%o17)                       12.19511843643877
```

Hence the total area enclosed is about 12.195. Maxima tries to calculate exactly, replacing floating point numbers with ratios of integers, and the default is to warn the user about these replacements. Hence we have used **ratprint : false$** to turn off this warning.

7.5 Arc Length of an Ellipse

Consider an ellipse whose equation is

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1. \tag{7.9}$$

in which we assume a is the semi-major axis of the ellipse so $a > b$. In the first quadrant ($0 \le x \le a$ and $0 \le y \le b$), we can solve for y as a function of x :

$$y(x) = b\sqrt{1 - (x/a)^2}. \tag{7.10}$$

If S is the arc length of the ellipse, then, by symmetry, one fourth of the arclength can be calculated using the first quadrant integral

$$\frac{S}{4} = \int_0^a \sqrt{1 + \left(\frac{dy}{dx}\right)^2}\, dx \tag{7.11}$$

We will start working with the argument of the square root, making a change of variable $x \to z$, with $x = a\,z$, so $dx = a\,dz$, and a will come outside the integral from the transformation of dx. The integration limits using the z variable are $0 \le z \le 1$.

We will then replace the semi-minor axis b by an expression depending on the ellipse eccentricity e, which has $0 < e \le 1$, and whose square is given by

$$e^2 = 1 - \left(\frac{b}{a}\right)^2 \le 1 \tag{7.12}$$

(since $b < a$), so

$$b = a\sqrt{1 - e^2} \tag{7.13}$$

```
(%i1) assume(a>0,b>0,a>b,e>0,e<1 )$
(%i2) y:b*sqrt(1 - (x/a)^2)$
(%i3) dydx : diff(y,x)$
```

```
(%i4) e1 : 1 + dydx^2;
                                 2  2
                                b  x
(%o4)                        ----------- + 1
                                     2
                              4     x
                             a  (1 - --)
                                     2
                                    a
(%i5) e2 : ( subst( [x = a*z,b = a*sqrt(1-e^2)],e1 ),ratsimp(%%) );
                                 2  2
                                e  z  - 1
(%o5)                           ---------
                                   2
                                  z  - 1
(%i6) e3 : (-num(e2))/(-denom(e2));
                                     2  2
                                1 - e  z
(%o6)                           ---------
                                      2
                                 1 - z
(%i7) e4 : dz*sqrt(num(e3))/sqrt(denom(e3));
                                         2  2
                           dz sqrt(1 - e  z )
(%o7)                      ------------------
                                         2
                              sqrt(1 - z )
```

The two substitutions give us expression **e2**, and we use a desperate device to multiply the top and bottom by (-1) to get **e3**. We then ignore the factor a which comes outside the integral and consider what is now inside the integral sign (with the required square root).

We now make another change of variables, with $z \to u$, $z = \sin(u)$, so $dz = \cos(u)\,du$. The lower limit of integration $z = 0 = \sin(u)$ transforms into $u = 0$, and the upper limit $z = 1 = \sin(u)$ transforms into $u = \pi/2$.

```
(%i8) e5 : subst( [z = sin(u), dz = cos(u)*du ],e4 );
                                              2    2
                        du cos(u) sqrt(1 - e  sin (u))
(%o8)                   ------------------------------
                                           2
                              sqrt(1 - sin (u))
```

We now use **trigsimp** but help Maxima out with an **assume** statement about $\cos(u)$ and $\sin(u)$.

```
(%i9) assume(cos(u)>0, sin(u) >0)$
(%i10) e6 : trigsimp(e5);
                                         2    2
(%o10)                     du sqrt(1 - e  sin (u))
```

We then have

$$\frac{S}{4} = a \int_0^{\pi/2} \sqrt{1 - e^2 \sin^2 u}\, du \tag{7.14}$$

Although **integrate** cannot return an expression for this integral in terms of elementary functions, in this form one is able to recognise the standard trigonometric form of the complete elliptic integral of the second kind, (a

function tabulated numerous places and also available via Maxima).

Let

$$\mathbf{E}(\mathbf{k}) = \mathbf{E}(\phi = \pi/2, \mathbf{k}) = \int_0^{\pi/2} \sqrt{1 - \mathbf{k}^2 \sin^2 \mathbf{u}}\, \mathbf{du} \tag{7.15}$$

be the definition of the complete elliptic integral of the second kind.

The "incomplete" elliptic integral of the second kind (with two arguments) is

$$\mathbf{E}(\phi, \mathbf{k}) = \int_0^{\phi} \sqrt{1 - \mathbf{k}^2 \sin^2 \mathbf{u}}\, \mathbf{du}. \tag{7.16}$$

Hence we have for the arc length of our ellipse

$$\mathbf{S} = 4\,\mathbf{a}\,\mathbf{E}(\mathbf{e}) = 4\,\mathbf{a}\,\mathbf{E}(\pi/2, \mathbf{e}) = 4\,\mathbf{a} \int_0^{\pi/2} \sqrt{1 - \mathbf{e}^2 \sin^2 \mathbf{u}}\, \mathbf{du} \tag{7.17}$$

We can evaluate numerical values using **elliptic_ec**, where
$\mathbf{E}(\mathbf{k}) = \mathrm{elliptic_ec}(\mathbf{k}^2)$, so $\mathbf{S} = 4\,\mathbf{a}\,\mathrm{elliptic_ec}(\mathbf{e}^2)$.

As a numerical example, take $\mathbf{a} = 3$, $\mathbf{b} = 2$, so $\mathbf{e}^2 = 5/9$, and

```
(%i11) float(12*elliptic_ec(5/9));
(%o11)                         15.86543958929059
```

We can check this numerical value using **`quad_qags`**:

```
(%i12) first( quad_qags(12*sqrt(1 - (5/9)*sin(u)^2),u,0,%pi/2) );
(%o12)                          15.86543958929059
```

7.6 Double Integrals and the Area of an Ellipse

Maxima has no core function which will compute a symbolic double definite integral, (although it would be easy to construct one). Instead of constructing such a homemade function, to evaluate the double integral

$$\int_{\mathbf{u1}}^{\mathbf{u2}} \mathbf{du} \int_{\mathbf{v1(u)}}^{\mathbf{v2(u)}} \mathbf{dv}\, \mathbf{f}(\mathbf{u}, \mathbf{v}) \equiv \int_{\mathbf{u1}}^{\mathbf{u2}} \left(\int_{\mathbf{v1(u)}}^{\mathbf{v2(u)}} \mathbf{f}(\mathbf{u}, \mathbf{v})\, \mathbf{dv} \right) \mathbf{du} \tag{7.18}$$

we use the Maxima code

```
integrate( integrate( f(u,v),v,v1(u),v2(u) ), u, u1,u2 )
```

in which $\mathbf{f}(\mathbf{u}, \mathbf{v})$ can either be an expression depending on the variables $\mathbf{u}$ and $\mathbf{v}$, or a Maxima function, and likewise $\mathbf{v1}(\mathbf{u})$ and $\mathbf{v2}(\mathbf{u})$ can either be expressions depending on $\mathbf{u}$ or Maxima functions. Both $\mathbf{u}$ and $\mathbf{v}$ are "dummy variables", since the value of the resulting double integral does not depend on our choice of symbols for the integration variables; we could just as well use $\mathbf{x}$ and $\mathbf{y}$.

Example 1: Area of a Unit Square

The area of a unit square (the sides have length 1) is:

$$\int_0^1 dx \int_0^1 dy \equiv \int_0^1 \left(\int_0^1 dy \right) dx \tag{7.19}$$

which is done in Maxima as:

```
(%i1) integrate( integrate(1,y,0,1), x,0,1 );
(%o1)                                1
```

Example 2: Area of an Ellipse

We seek the area of the ellipse such that points (x, y) on the boundary must satisfy:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1. \tag{7.20}$$

The area will be four times the area of the first quadrant. The "first quadrant" refers to the region $0 \le x \le a$ and $0 \le y \le b$. For a given value of $y > 0$, the region inside the arc of the ellipse in the first quadrant is determined by $0 \le x \le x_{max}$, where $x_{max} = (a/b)\sqrt{b^2 - y^2}$. For a given value of $x > 0$, the region inside the arc of the ellipse in the first quadrant is determined by $0 \le y \le y_{max}$, where $y_{max} = (b/a)\sqrt{a^2 - x^2}$.

One way to find the area of the first quadrant of this ellipse is to sum the values of $dA = dx\,dy$ by "rows", fixing y and summing over x from 0 to x_{max} (which depends on the y chosen). That first sum over the x coordinate accumulates the area of that row, located at y and having width dy. To get the total area (of the first quadrant) we then sum over rows, by letting y vary from 0 to b.
This method corresponds to the formula

$$\frac{A}{4} = \int_0^b \left(\int_0^{(a/b)\sqrt{b^2-y^2}} dx \right) dy. \tag{7.21}$$

Here we calculate the first quadrant area using this method of summing over the area of each row.

```
(%i1) facts();
(%o1)                               []
(%i2) assume(a > 0, b > 0, x > 0, x < a, y > 0,y < b )$
(%i3) facts();
(%o3)          [a > 0, b > 0, x > 0, a > x, y > 0, b > y]
(%i4) [xmax : (a/b)*sqrt(b^2 - y^2),ymax : (b/a)*sqrt(a^2-x^2)]$
(%i5) integrate( integrate( 1,x,0,xmax), y,0,b );
                              %pi a b
(%o5)                         -------
                                 4
```

which implies that the area interior to the complete ellipse is $\pi\, a\, b$.

Note that we have tried to be "overly helpful" to Maxima's **integrate** function by constructing an "assume list" with everything we can think of about the variables and parameters in this problem. The main advantage of doing this is to reduce the number of questions which **integrate** decides it has to ask the user.

You might think that **integrate** would not ask for the sign of $(y - b)$, or the sign of $(x - a)$, since it should infer that sign from the integration limits. However, the **integrate** algorithm is "super cautious" in trying to never present you with a wrong answer. The general philosophy is that the user should be willing to work with **integrate** to assure a correct answer, and if that involves answering questions, then so be it.

The second way to find the area of the first quadrant of this ellipse is to sum the values of $dA = dx\,dy$ by "columns", fixing x and summing over y from 0 to y_{max} (which depends on the x chosen). That first sum over the y coordinate accumulates the area of that column, located at x and having width dx. To get the total area (of the first quadrant) we then sum over columns, by letting x vary from 0 to a.

This method corresponds to the formula

$$\frac{A}{4} = \int_0^a \left(\int_0^{(b/a)\sqrt{a^2 - x^2}} dy \right) dx \tag{7.22}$$

and is implemented by

```
(%i6) integrate( integrate( 1,y,0,ymax), x,0,a );
                                        %pi a b
(%o6)                                   -------
                                           4
```

Example 3: Moment of Inertia for Rotation about the x-axis

We next calculate the moment of inertia for rotation of an elliptical lamina (having semi-axes a, b) about the x axis. We will call this quantity I_x. Each small element of area $dA = dx\,dy$ has a mass dm given by $\sigma\,dA$, where σ is the mass per unit area, which we assume is a constant independent of where we are on the lamina, and which we can express in terms of the total mass m of the elliptical lamina, and the total area $A = \pi\,a\,b$ as

$$\sigma = \frac{\text{total mass}}{\text{total area}} = \frac{m}{\pi\,a\,b} \tag{7.23}$$

Each element of mass dm contributes an amount $y^2\,dm$ to the moment of inertia I_x, where y is the distance of the mass element from the x axis. The complete value of I_x is then found by summing this quantity over the whole elliptical laminate, or because of the symmetry, by summing this quantity over the first quadrant and multiplying by 4.

$$I_x = \int\int_{\text{ellipse}} y^2\,dm = \int\int_{\text{ellipse}} y^2\,\sigma\,dx\,dy = 4\,\sigma\,\int\int_{\text{first quadrant}} y^2\,dx\,dy \tag{7.24}$$

Here we use the method of summing over rows:

```
(%i7) sigma : m/(%pi*a*b)$
(%i8) 4*sigma*integrate( integrate(y^2,x,0,xmax),y,0,b );
                                          2
                                         b  m
(%o8)                                    ----
                                          4
```

Hence we have derived $I_x = m\,b^2/4$ for the moment of inertia of an elliptical laminate having semi-axes a, b for rotation about the x axis.

Finally, let's use **forget** to remove our list of assumptions, to show you the types of questions which can arise. We try calculating the area of the first quadrant, using the method of summing over rows, without any assumptions provided:

```
(%i9) forget(a > 0, b > 0, x > 0, x < a, y > 0,y < b )$
(%i10) facts();
(%o10)                          []
(%i11) integrate( integrate(1,x,0,xmax),y,0,b);
Is  (y - b) (y + b)  positive, negative, or zero?
n;
     2    2
Is  b  - y   positive or zero?
p;
Is  b  positive, negative, or zero?
p;
                                      %pi a b
(%o11)                                -------
                                         4
```

and if we just tell Maxima that **b** is positive:

```
(%i12) assume(b>0)$
(%i13) integrate( integrate(1,x,0,xmax),y,0,b);
Is  (y - b) (y + b)  positive, negative, or zero?
n;
     2    2
Is  b  - y   positive or zero?
p;
                                      %pi a b
(%o13)                                -------
                                         4
```

This may give you some feeling for the value of providing some help to **integrate**.

7.7 Triple Integrals: Volume and Moment of Inertia of a Solid Ellipsoid

Consider an ellipsoid with semi-axes (a, b, c) such that $-a \le x \le a$, $-b \le y \le b$, and $-c \le z \le c$. We also assume here that $a > b > c$. Points (x, y, z) which live on the surface of this ellipsoid satisfy the equation of the surface

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1. \tag{7.25}$$

Volume

The volume of this ellipsoid will be 8 times the volume of the first octant, defined by $0 \le x \le a$, $0 \le y \le b$, and $0 \le z \le c$. To determine the volume of the first octant we pick fixed values of (x, y) somewhere in the first octant, and sum the elementary volume $dV(x, y, z) = dx\,dy\,dz$ over all accessible values of z in this first octant, $0 \le z \le c\sqrt{1 - (x/a)^2 - (y/b)^2}$. The result will still be proportional to $dx\,dy$, and we sum the z-direction cylinders over the entire range of y (accessible in the first octant), again holding x fixed as before, so for given x, we have $0 \le y \le b\sqrt{1 - (x/a)^2}$. We now have a result proportional to dx (a sheet of thickness dx whose normal is in the direction of the x axis) which we sum over all values of x accessible in the first octant (with no restrictions): $0 \le x \le a$. Thus we need to evaluate the triple integral

$$\frac{V}{8} = \int_0^a dx \int_0^{ymax(x)} dy \int_0^{zmax(x,y)} dz \equiv \int_0^a \left[\int_0^{ymax(x)} \left(\int_0^{zmax(x,y)} dz \right) dy \right] dx \tag{7.26}$$

Here we call on **integrate** to do these integrals.

```
(%i1) assume(a>0,b>0,c>0,a>b,a>c,b>c)$
(%i2) assume(x>0,x<a,y>0,y<b,z>0,z<c)$
(%i3) zmax:c*sqrt(1-x^2/a^2-y^2/b^2)$
(%i4) ymax:b*sqrt(1-x^2/a^2)$
(%i5) integrate( integrate( integrate(1,z,0,zmax),y,0,ymax),x,0,a );
      2  2    2  2    2  2
Is  a  y  + b  x  - a  b   positive, negative, or zero?

n;
                                   %pi a b c
(%o5)                              ---------
                                       6
(%i6) vol : 8*%;
                                  4 %pi a b c
(%o6)                             -----------
                                       3
```

To answer the sign question posed by Maxima, we can look at the special case $y = 0$ and note that since $x^2 \le a^2$, the expression will almost always be negative (ie., except for points of zero measure). Hence the expression is negative for all points interior to the surface of the ellipsoid. We thus have the result that the volume of an ellipsoid having semi-axes (a, b, c) is given by

$$V = \frac{4\,\pi}{3} a\,b\,c \tag{7.27}$$

We can now remove the assumption $a > b > c$, since what we call the x axis is up to us and we could have chosen any of the principal axis directions of the ellipsoid the x axis.

Of course we will get the correct answer if we integrate over the total volume:

```
(%i7) [zmin:-zmax,ymin:-ymax]$
(%i8) integrate( integrate( integrate(1,z,zmin,zmax),y,ymin,ymax),x,-a,a );
      2  2    2  2    2  2
Is  a  y  + b  x  - a  b   positive, negative, or zero?

n;
                                  4 %pi a b c
(%o8)                             -----------
                                       3
```

Moment of Inertia

If each small volume element $dV = dx\,dy\,dz$ has a mass given by $dm = \rho\,dV$, where ρ is the mass density, and if ρ is a constant, then it is easy to calculate the moment of inertia I_3 for rotation of the ellipsoid about the z axis:

$$I_3 = \int\int\int (x^2 + y^2)\,dm = \rho \int\int\int (x^2 + y^2)\,dx\,dy\,dz \tag{7.28}$$

where the integration is over the volume of the ellipsoid. The constant mass density ρ is the mass of the ellipsoid divided by its volume.

Here we define that constant density in terms of our previously found volume **vol** and the mass **m**, and proceed to calculate the moment of inertia:

```
(%i9) rho:m/vol;
                                        3 m
(%o9)                               -----------
                                    4 %pi a b c
(%i10) i3:rho*integrate(integrate(integrate(x^2+y^2,z,zmin,zmax),
                       y,ymin,ymax),x,-a,a );
     2  2    2  2    2  2
Is  a  y  + b  x  - a  b   positive, negative, or zero?

n;
                                   5  3          7
                         (8 %pi a  b  + 8 %pi a  b) m
(%o10)                   ----------------------------
                                           5
                                 40 %pi a  b
(%i11) ratsimp(i3);
                                      2    2
                                    (b  + a ) m
(%o11)                              -----------
                                         5
```

Hence the moment of inertia for the rotation about the z axis of a solid ellipsoid having mass m, and semi-axes (a, b, c) is:

$$I_3 = m\,(a^2 + b^2)/5. \tag{7.29}$$

7.8 Derivative of a Definite Integral with Respect to a Parameter

Consider a definite integral in which the dummy integration variable is x and the integrand $f(x, y)$ is a function of both x and some parameter y. We also assume that the limits of integration (a, b) are also possibly functions of the parameter y. You can find the proof of the following result (which assumes some mild restrictions on the funtions $f(x, y)$, $a(y)$ and $b(y)$) in calculus texts:

$$\frac{d}{dy}\int_{a(y)}^{b(y)} f(x, y)\,dx = \int_a^b \frac{\partial f(x, y)}{\partial y}\,dy + f(b(y), y)\,\frac{d\,b}{d\,y} - f(a(y), y)\,\frac{d\,a}{d\,y} \tag{7.30}$$

Here we ask Maxima for this result for arbitrary functions:

```
(%i1) expr : 'integrate(f(x,y),x,a(y),b(y) );
                                b(y)
                               /
                               [
(%o1)                          I     f(x, y) dx
                               ]
                               /
                                a(y)
(%i2) diff(expr,y);
                                                           b(y)
                                                          /
                 d                           d            [      d
(%o2) f(b(y), y) (-- (b(y))) - f(a(y), y) (-- (a(y))) + I     (-- (f(x, y))) dx
                 dy                          dy           ]      dy
                                                          /
                                                           a(y)
```

and we see that Maxima assumes we have enough smoothness in the functions involved to write down the formal answer in the correct form.

Example 1

We next display a simple example and begin with the simplest case, which is that the limits of integration do not depend on the parameter y.

```
(%i3) expr : 'integrate(x^2 + 2*x*y, x,a,b);
                                b
                               /
                               [              2
(%o3)                          I   (2 x y + x ) dx
                               ]
                               /
                                a
(%i4) diff(expr,y);
                                      b
                                     /
                                     [
(%o4)                              2 I   x dx
                                     ]
                                     /
                                      a
(%i5) (ev(%,nouns), ratsimp(%%) );
                                     2    2
(%o5)                               b  - a
(%i6) ev(expr,nouns);
                               2      3      2      3
                            3 b  y + b    3 a  y + a
(%o6)                       ----------- - -----------
                                 3             3
(%i7) diff(%,y);
                                     2    2
(%o7)                               b  - a
```

In the last two steps, we have verified the result by first doing the original integral and then taking the derivative.

Example 2

As a second example, we use an arbitrary upper limit $b(y)$, and then evaluate the resulting derivative expression for $b(y) = y^2$.

```
(%i1) expr : 'integrate(x^2 + 2*x*y, x,a,b(y) );
                              b(y)
                             /
                             [               2
(%o1)                        I     (2 x y + x ) dx
                             ]
                             /
                              a
```

```
(%i2) diff(expr,y);
                                                     b(y)
                                                    /
                  2                 d               [
(%o2)           (b (y) + 2 y b(y)) (-- (b(y))) + 2 I     x dx
                                    dy              ]
                                                    /
                                                     a
(%i3) ( ev(%,nouns,b(y)=y^2 ), expand(%%) );
                                 5     4    2
(%o3)                         2 y  + 5 y  - a
(%i4) ev(expr,nouns);
                        3           2          2      3
                       b (y) + 3 y b (y)   3 a  y + a
(%o4)                  ----------------- - -----------
                               3                3
(%i5) diff(%,y);
           2    d                           d                 2
        3 b (y) (-- (b(y))) + 6 y b(y) (-- (b(y))) + 3 b (y)
                dy                          dy                       2
(%o5)   --------------------------------------------------- - a
                                   3
(%i6) ( ev(%,nouns,b(y)=y^2), (expand(%%) );
                                 5     4    2
(%o6)                         2 y  + 5 y  - a
```

We have again done the differentiation two ways as a check on consistency.

Example 3

An easy example is to derive

$$\frac{d}{dt}\int_{t}^{t^2}(2\,x+t)\,d\,x = 4\,t^3+3\,t^2-4\,t \tag{7.31}$$

```
(%i7) expr : 'integrate(2*x + t,x,t,t^2);
                                  2
                                 t
                                /
                                [
(%o7)                           I   (2 x + t) dx
                                ]
                                /
                                 t
(%i8) (diff(expr,t),expand(%%) );
                                  3     2
(%o8)                          4 t  + 3 t  - 4 t
```

Example 4

Here is an example which shows a common use of the differentiation of an integral with respect to a parameter. The integral

$$f_1(a, w) = \int_0^{\infty} x\, e^{-a\,x} \cos(w\,x)\, dx \tag{7.32}$$

can be done by Maxima with no questions asked, if we tell Maxima that $a > 0$ and $w > 0$.

```
(%i1) assume( a > 0, w > 0 )$
(%i2) integrate(x*exp(-a*x)*cos(w*x),x,0,inf);
                                         2    2
                                        w  - a
(%o2)                            - -----------------
                                    4      2  2    4
                                   w  + 2 a  w  + a
```

But if we could not find this result directly as above, we could find the result by setting $f_1(a, w) = -\partial f_2(a, w)/(\partial a)$, where

$$f_2(a, w) = \int_0^{\infty} e^{-a\,x} \cos(w\,x)\, dx \tag{7.33}$$

since the differentiation will result in the integrand being multiplied by the factor $(-x)$ and produce the negative of the integral of interest.

```
(%i3) i1 : 'integrate(exp(-a*x)*cos(w*x),x,0,inf)$
(%i4) i2 : ev(i1,nouns);
                                          a
(%o4)                                  -------
                                        2    2
                                       w  + a
(%i5) di2 : (diff(i2,a), ratsimp(%%) );
                                        2    2
                                       w  - a
(%o5)                              -----------------
                                    4      2  2    4
                                   w  + 2 a  w  + a
(%i6) result : (-1)*(diff(i1,a) = di2 );
                inf
               /                                   2    2
               [        - a x                     w  - a
(%o6)          I     x %e      cos(w x) dx = - -----------------
               ]                                4      2  2    4
               /                               w  + 2 a  w  + a
                0
```

which reproduces the original result returned by **integrate**.

7.9 Integration by Parts

Suppose $\mathbf{f(x)}$ and $\mathbf{g(x)}$ are two continuously differentiable functions in the interval of interest. Then the **integration by parts rule** states that given an interval with endpoints $(\mathbf{a}, \mathbf{b})$, (and of course assuming the derivatives exist) one has

$$\int_{a}^{b} f(x)\, g'(x)\, dx = [f(x)\, g(x)]_{a}^{b} - \int_{a}^{b} f'(x)\, g(x)\, dx \tag{7.34}$$

where the prime indicates differentiation. This result follows from the product rule of differentiation. This rule is often stated in the context of indefinite integrals as

$$\int f(x)\, g'(x)\, dx = f(x)\, g(x) - \int f'(x)\, g(x)\, dx \tag{7.35}$$

or in an even shorter form, with $\mathbf{u} = \mathbf{f(x)}$, $\mathbf{du} = \mathbf{f'(x)\, dx}$, $\mathbf{v} = \mathbf{g(x)}$, and $\mathbf{dv} = \mathbf{g'(x)\, dx}$, as

$$\int u\, dv = u\, v - \int v\, du \tag{7.36}$$

In practice, we are confronted with an integral whose integrand can be viewed as the product of two factors, which we will call $\mathbf{f(x)}$ and $\mathbf{h(x)}$:

$$\int f(x)\, h(x)\, dx \tag{7.37}$$

and we wish to use integration by parts to get an integral involving the derivative of the first factor, $\mathbf{f(x)}$, which will hopefully result in a simpler integral. We then identify $\mathbf{h(x)} = \mathbf{g'(x)}$ and solve this equation for $\mathbf{g(x)}$ (by integrating; this is also a choice based on the ease of integrating the second factor $\mathbf{h(x)}$ in the given integral). Having $\mathbf{g(x)}$ in hand we can then write out the result using the indefinite integral integration by parts rule above. We can formalize this process for an *indefinite integral* with the Maxima code:

```
(%i1) iparts(f,h,var):= block([g ],
        g : integrate(h,var),
        f*g - 'integrate(g*diff(f,var),var ) )$
```

Let's practice with the integral $\int \mathbf{x^2 \sin(x)\, dx}$, in which $\mathbf{f(x)} = \mathbf{x^2}$ and $\mathbf{h(x)} = \mathbf{\sin(x)}$, so we need to be able to integrate $\sin(x)$ and want to transfer a derivative on to $\mathbf{x^2}$, which will reduce the first factor to $\mathbf{2\, x}$. Notice that it is usually easier to work with "Maxima expressions" rather than with "Maxima functions" in a problem like this.

```
(%i2) iparts(x^2,sin(x),x);
                              /
                              [                 2
(%o2)                      2 I x cos(x) dx - x  cos(x)
                              ]
                              /
(%i3) (ev(%,nouns),expand(%%) );
                                      2
(%o3)                   2 x sin(x) - x  cos(x) + 2 cos(x)
(%i4) collectterms(%,cos(x) );
                                             2
(%o4)                     2 x sin(x) + (2 - x ) cos(x)
```

If we were not using Maxima, but doing everything by hand, we would use two integrations by parts (in succession) to remove the factor $\mathbf{x^2}$ entirely, reducing the original problem to simply knowing the integrals of $\sin(x)$ and $\cos(x)$.

Of course, with an integral as simple as this example, there is no need to help Maxima out by integrating by parts.

```
(%i5) integrate(x^2*sin(x),x);
                                                2
(%o5)                       2 x sin(x) + (2 - x ) cos(x)
```

We can write a similar Maxima function to transform definite integrals via integration by parts.

```
(%i6) idefparts(f,h,var,v1,v2):= block([g ],
         g : integrate(h,var),
           'subst(v2,var, f*g)  - 'subst(v1,var, f*g ) -
               'integrate(g*diff(f,var),var,v1,v2 ) )$
(%i7) idefparts(x^2,sin(x),x,0,1);
          1
         /
         [                                  2
(%o7) 2 I   x cos(x) dx + substitute(1, x, - x  cos(x))
         ]
         /
          0
                                                                        2
                                                 - substitute(0, x, - x  cos(x))
(%i8) (ev(%,nouns),expand(%%) );
(%o8)                         2 sin(1) + cos(1) - 2
(%i9) integrate(x^2*sin(x),x,0,1);
(%o9)                         2 sin(1) + cos(1) - 2
```

7.10 Change of Variable and changevar

Many integrals can be evaluated most easily by making a change of variable of integration. A simple example is:

$$\int 2\,x\,(x^2+1)^3\,dx = \int (x^2+1)^3\,d\,(x^2+1) = \int u^3\,du = u^4/4 = (x^2+1)^4/4 \tag{7.38}$$

There is a function in Maxima, called **changevar** which will help you change variables in a one-dimensional integral (either indefinite or definite). However, this function is buggy at present and it is safer to do the change of variables "by hand".

Function: **changevar(integral-expression, g(x,u), u, x)**
Makes the change of variable in a given *noun form* **integrate** expression such that the old variable of integration is x, the new variable of integration is u, and x and u are related by the equation $g(x,u) = 0$.

Example 1

Here we use this Maxima function on the simple indefinite integral $\int 2\,x\,(x^2+1)^3\,dx$ we have just done "by hand":

```
(%i1) expr : 'integrate(2*x*(x^2+1)^3,x);
                                   /
                                   [      2     3
(%o1)                            2 I x (x  + 1)   dx
                                   ]
                                   /
(%i2) changevar(expr,x^2+1-u,u,x);
                                       /
                                       [  3
(%o2)                                  I u  du
                                       ]
                                       /
(%i3) ev(%, nouns);
                                          4
                                         u
(%o3)                                    --
                                         4
(%i4) ratsubst(x^2+1,u,%);
                          8      6      4      2
                         x  + 4 x  + 6 x  + 4 x  + 1
(%o4)                    ---------------------------
                                      4
(%i5) subst(x^2+1,u,%o3);
                                    2     4
                                  (x  + 1)
(%o5)                             ---------
                                      4
(%i6) subst(u=(x^2+1),%o3);
                                    2     4
                                  (x  + 1)
(%o6)                             ---------
                                      4
```

The original indefinite integral is a function of x, which we obtain by replacing u by its equivalent as a function of x. We have shown three ways to make this replacement to get a function of x, using **subst** and **ratsubst**.

Example 2

As a second example, we use a change of variables to find $\int\,\left[(x+2)/\sqrt{x+1}\right]\,dx$.

```
(%i7) expr : 'integrate( (x+2)/sqrt(x+1),x);
                                  /
                                  [     x + 2
(%o7)                             I ----------- dx
                                  ] sqrt(x + 1)
                                  /
```

```
(%i8) changevar(expr,u - sqrt(x+1),u,x);
Is  u  positive, negative, or zero?

pos;
                                 /
                                 [      2
(%o8)                            I  (2 u  + 2) du
                                 ]
                                 /
(%i9) ev(%, nouns);
                                      3
                                   2 u
(%o9)                              ---- + 2 u
                                    3
(%i10) subst(u = sqrt(x+1),%);
                                   3/2
                          2 (x + 1)
(%o10)                    ------------ + 2 sqrt(x + 1)
                               3
```

Of course Maxima can perform the original integral in these two examples without any help, as in:

```
(%i11) integrate((x+2)/sqrt(x+1),x);
                                      3/2
                             (x + 1)
(%o11)                    2 (---------- + sqrt(x + 1))
                                 3
```

However, there are occasionally cases in which you can help Maxima find an integral (for which Maxima can only return the noun form) by first making your own change of variables and then letting Maxima try again.

Example 3

Here we use **changevar** with a definite integral, using the same integrand as in the previous example. For a definite integral, the variable of integration is a "dummy variable", and the result is not a function of that dummy variable, so there is no issue about replacing the new integration variable **u** by the original variable **x** in the result.

```
(%i12) expr : 'integrate( (x+2)/sqrt(x+1),x,0,1);
                                 1
                                /
                                [      x + 2
(%o12)                          I  ----------- dx
                                ]  sqrt(x + 1)
                                /
                                 0
```

```
(%i13) changevar(expr,u - sqrt(x+1),u,x);
Is  u  positive, negative, or zero?

pos;
                             sqrt(2)
                            /
                            [            2
(%o13)                      I        (2 u  + 2) du
                            ]
                            /
                             1
(%i14) ev(%, nouns);
                                10 sqrt(2)   8
(%o14)                          ---------- - -
                                    3        3
```

Example 4

We next discuss an example which shows that one needs to pay attention to the possible introduction of obvious sign errors when using **changevar**. The example is the evaluation of the definite integral $\int_0^1 e^{y^2}\,dy$, in which y is a real variable. Since the integrand is a positive (real) number over the interval $0 < y \leq 1$, the definite integral must be a positive (real) number. The answer returned directly by Maxima's **integrate** function is:

```
(%i1) fpprintprec:8$
(%i2) i1:integrate (exp (y^2),y,0,1);
                              sqrt(%pi) %i erf(%i)
(%o2)                       - --------------------
                                       2
```

and **float** yields

```
(%i3) float(i1);
(%o3)                        - 0.886227 %i erf(%i)
```

so we see if we can get a numerical value from **erf(%i)** by multiplying **%i** by a floating point number:

```
(%i4) erf(1.0*%i);
(%o4)                             1.6504258 %i
```

so to get a numerical value for the integral we use the same trick

```
(%i5) float(subst(1.0*%i,%i,i1));
(%o5)                               1.4626517
```

Maxima's symbol **erf(z)** represents the error function $\mathrm{Erf}(z)$. We have discussed the Maxima function **erf(x)** for real x in Example 4 in Sec.(7.2). Here we have a definite integral result returned in terms of **erf(%i)**, which is the error function with a pure imaginary agument and we have just seen that **erf(%i)** is purely imaginary with an approximate value **1.65*%i**.

We can confirm the numerical value of the integral **i1** using the quadrature routine **quad_qags**:

```
(%i6) quad_qags(exp(y^2),y,0,1);
(%o6)                 [1.4626517, 1.62386965E-14, 21, 0]
```

and we see agreement.

Let's ask Maxima to change variables in this definite integral from y to $u = y^2$ in the following way:

```
(%i7) expr : 'integrate(exp(y^2),y,0,1);
                                    1
                                   /     2
                                   [    y
(%o7)                              I  %e    dy
                                   ]
                                   /
                                    0
(%i8) changevar(expr,y^2-u,u,y);
                                    1
                                   /      u
                                   [    %e
                                   I  ------- du
                                   ]  sqrt(u)
                                   /
                                    0
(%o8)                            - -------------
                                         2
(%i9) ev(%, nouns);
                              sqrt(%pi) %i erf(%i)
(%o9)                         --------------------
                                       2
(%i10) float(subst(1.0*%i,%i,%));
(%o10)                             - 1.4626517
```

which is the negative of the correct result. Evidently, Maxima uses **solve**$(y^2 = u,\ y)$ to find $y(u)$ and even though there are two solutions $y = \pm\sqrt{u}$, Maxima picks the wrong solution without asking the user a clarifying question. **We need to force Maxima to use the correct relation** between y and u, as in:

```
(%i11) changevar(expr,y-sqrt(u),u,y);
Is  y  positive, negative, or zero?

pos;
                                   1
                                  /      u
                                  [    %e
                                  I  ------- du
                                  ]  sqrt(u)
                                  /
                                   0
(%o11)                            -------------
                                        2
(%i12) ev(%, nouns);
                               sqrt(%pi) %i erf(%i)
(%o12)                       - --------------------
                                        2
(%i13) float(subst(1.0*%i,%i,%));
(%o13)                              1.4626517
```

which is now the correct result with the correct sign.

Example 5

We now discuss an example of a change of variable in which **changevar** produces the wrong overall sign, even though we try to be very careful. We consider the indefinite integral $\int \left(x/\sqrt{x^2 - 4} \right) dx$, which **integrate** returns as:

```
(%i1) integrate(x/sqrt(x^2-4),x);
                                        2
(%o1)                           sqrt(x   - 4)
```

Now consider the change of variable $x \to t$ with $x = 2/\cos(t)$.

We will show first the **changevar** route (with its error) and then how to do the change of variables "by hand", but with Maxima'a assistance. Here we begin with assumptions about the variables involved.

```
(%i2) assume(x > 2, t > 0, t < 1.5, cos(t) > 0, sin(t) > 0 );
(%o2)           [x > 2, t > 0, t < 1.5, cos(t) > 0, sin(t) > 0]
(%i3) nix : 'integrate(x/sqrt(x^2-4),x);
                               /
                               [      x
(%o3)                          I ------------ dx
                               ]       2
                               / sqrt(x   - 4)
(%i4) nixt : changevar(nix,x-2/cos(t), t, x) ;
                    /
                    [                  sin(t)
(%o4)        - 2 %i I ----------------------------------------- dt
                    ]                     2
                    / sqrt(cos(t) - 1) cos (t) sqrt(cos(t) + 1)
(%i5) nixt : rootscontract(nixt);
                            /
                            [          sin(t)
(%o5)                  - 2 %i I ------------------------- dt
                            ]    2              2
                            / cos (t) sqrt(cos (t) - 1)
(%i6) nixt : scanmap('trigsimp,nixt);
                                   /
                                   [    1
(%o6)                            - 2 I ------- dt
                                   ]    2
                                   / cos (t)
(%i7) ev(nixt,nouns);
(%o7)                               - 2 tan(t)
```

Since we have assumed $t > 0$, we have $\tan(t) > 0$, so **changevar** is telling us the indefinite integral is a negative number for the range of t assumed.

Since we are asking for an indefinite integral, and we want the result in terms of the original variable x, we would need to do some more work on this answer, maintaing the assumptions we have made. We will do that work after we have repeated this change of variable, doing it "by hand".

We work on the product $f(x)\,dx$:

```
(%i8) ix : subst(x=2/cos(t),x/sqrt(x^2 - 4) )* diff(2/cos(t));
                                4 sin(t) del(t)
(%o8)                      -------------------------
                                   4             3
                           sqrt(------- - 4) cos (t)
                                   2
                                cos (t)
(%i9) ix : trigsimp(ix);
                                   2 del(t)
(%o9)                           - -----------
                                      2
                                   sin (t) - 1
(%i10) ix : ratsubst(1,cos(t)^2+sin(t)^2,ix);
                                   2 del(t)
(%o10)                              --------
                                      2
                                   cos (t)
(%i11) integrate(coeff(ix,del(t) ) ,t);
(%o11)                             2 tan(t)
```

which is the result **changevar** should have produced. Now let's show how we can get back to the indefinite integral produced the direct use of **integrate**.

```
(%i12) subst(tan(t)= sqrt(sec(t)^2-1),2*tan(t) );
                                        2
(%o12)                        2 sqrt(sec (t) - 1)
(%i13) subst(sec(t)=1/cos(t),%);
                                        1
(%o13)                        2 sqrt(------- - 1)
                                        2
                                     cos (t)
(%i14) subst(cos(t)=2/x,%);
                                        2
                                       x
(%o14)                          2 sqrt(-- - 1)
                                       4
(%i15) ratsimp(%);
                                       2
(%o15)                           sqrt(x  - 4)
```

This concludes our discussion of a change of variable of integration and our discussion of symbolic integration.

Maxima by Example:
Ch.8: Numerical Integration *

Edwin L. Woollett

September 16, 2010

Contents

8.1 Introduction 3
8.2 Numerical Integration Basic Tools: **quad_qags**, **romberg**, **quad_qagi** 3
  8.2.1 Syntax for Quadpack Functions 3
  8.2.2 Ouput List of Quadpack Functions and Error Code Values 3
  8.2.3 Integration Rule Parameters and Optional Arguments 4
  8.2.4 **quad_qags** for a Finite Interval 4
  8.2.5 **romberg** for a Finite Interval 7
  8.2.6 **quad_qags** and **romberg** for Numerical Double Integrals 8
  8.2.7 **quad_qagi** for an Infinite or Semi-infinite Interval 10
8.3 Numerical Integration: Sharper Tools 12
  8.3.1 **quad_qag** for a General Oscillatory Integrand 12
  8.3.2 **quad_qawo** for Fourier Series Coefficients 14
  8.3.3 **quad_qaws** for End Point Algebraic and Logarithmic Singularities 15
  8.3.4 **quad_qawc** for a Cauchy Principal Value Integral 17
  8.3.5 **quad_qawf** for a Semi-Infinite Range Cosine or Sine Fourier Transform 19
8.4 Finite Region of Integration Decision Tree 20
8.5 Semi-infinite or Infinite Region of Integration Decision Tree 21

*This version uses **Maxima 5.18.1**. This is a live document. Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions to `woollett@charter.net`

Preface

```
COPYING AND DISTRIBUTION POLICY
This document is part of a series of notes titled
"Maxima by Example" and is made available
via the author's webpage http://www.csulb.edu/~woollett/
to aid new users of the Maxima computer algebra system.

NON-PROFIT PRINTING AND DISTRIBUTION IS PERMITTED.

You may make copies of this document and distribute them
to others as long as you charge no more than the costs of printing.

These notes (with some modifications) will be published in book form
eventually via Lulu.com in an arrangement which will continue
to allow unlimited free download of the pdf files as well as the option
of ordering a low cost paperbound version of these notes.
```

Feedback from readers is the best way for this series of notes to become more helpful to new users of Maxima. *All* comments and suggestions for improvements will be appreciated and carefully considered.

```
The Maxima session transcripts were generated using the Xmaxima
graphics interface on a Windows XP computer, and copied into
a fancy verbatim environment in a latex file which uses the
fancyvrb and color packages.

We use qdraw.mac (available on the author's web page with Ch. 5 materials)
for plots.
```

```
  Maxima, a Computer Algebra System.
  Some numerical results depend on the Lisp version used.
  This chapter uses Version 5.18.1 (2009) using Lisp GNU
  Common Lisp (GCL) GCL 2.6.8 (aka GCL).
  http://maxima.sourceforge.net/
```

The author would like to thank the Maxima developers for their friendly help via the Maxima mailing list.

8.1 Introduction

This chapter is divided into four sections.

We first discuss the most needed and used tools for routine numerical integration. (We presented many examples of symbolic integration methods in Ch. 7). We then discuss more exotic tools you may be able to use once in a while. In the third short section we present a finite interval integration "decision tree". In the fourth short section we present an "infinite interval integration decision tree". In Chapter 9, we will discuss bigfloats and some examples of using Maxima for high precision quadrature. Chapter 10 covers the calculation of Fourier series coefficients and both Fourier transform and Laplace transform type integrals. Chapter 11 presents tools for the use of fast Fourier transforms with examples of use.

8.2 Numerical Integration Basic Tools: quad_qags, romberg, quad_qagi

8.2.1 Syntax for Quadpack Functions

All the Quadpack functions ("Q" is for "quadrature"), except two, are called using the syntax:

```
quad_qaxx ( expr, var, a, b, (other-required-args), optional-args)
```

The exceptions are the Cauchy principle value integration routine **quad_qawc** which does the integral $\int_a^b f(x)/(x-c)\,dx$, using the syntax `quad_qawc(expr,var,c,a,b,optional-args)`, and the cosine or sine Fourier transform integration routine **quad_qawf** which does integrals $\int_a^\infty f(x)\cos(\omega\, x)\,dx$ or $\int_a^\infty f(x)\sin(\omega\, x)\,dx$ using the syntax `quad_qawf(expr,var,a,omega,trig,optional-args)`.

8.2.2 Ouput List of Quadpack Functions and Error Code Values

All Quadpack functions return a **list** of four elements:

```
[integral.value, est.abs.error, num.integrand.evaluations, error.code].
```

The **error.code** (the fourth element of the returned list) can have the values:

0 - no problems were encountered

1 - too many sub-intervals were done

2 - excessive roundoff error is detected

3 - extremely bad integrand behavior occurs

4 - failed to converge

5 - integral is probably divergent or slowly convergent

6 - the input is invalid

8.2.3 Integration Rule Parameters and Optional Arguments

All the Quadpack functions (except one) include **epsrel**, **epsabs**, and **limit** as possible optional types of allowed arguments to override the default values of these parameters: **epsrel = 1e-8, epsabs = 0.0, limit = 200**.
To override the **epsrel** default value, for example, you would add the optional argument **epsrel = 1e-6** or **epsrel=1d-6** (both have the same effect).

These Quadpack functions apply an integration rule adaptively until an estimate of the integral of **expr** over the interval **(a, b)** is achieved within the desired absolute and relative error limits, **epsabs** and **epsrel**. With the default values **epsrel = 1e-8, epsabs = 0.0**, only **epsrel** plays a role in determining the convergence of the integration rules used, and this corresponds to getting the estimated relative error of the returned answer smaller than **epsrel**.

If you override the defaults with the two optional arguments (in any order) **epsrel = 0.0, epsabs = 1e-8**, for example, the value of **epsrel** will be ignored and convergence will have been achieved if the estimated absolute error is less than **epsabs**.

The integration region is divided into subintervals, and on each iteration the subinterval with the largest estimated error is bisected. This "adaptive method" reduces the overall error rapidly, as the subintervals become concentrated around local difficulties in the integrand.

The Quadpack parameter **limit**, whose default value is **200**, is the maximum number of subintervals to be used in seeking a convergent answer. To increase that limit, you would insert **limit=300**, for example.

8.2.4 quad_qags for a Finite Interval

The "s" on the end of **qags** is a signal that **quad_qags** has extra abilities in dealing with functions which have integrable **singularities**, but even if your function has no singular behavior, **quad_qags** is still the best choice due to the sophistication of the quadrature algorithm used. Use the syntax

```
quad_qags ( expr, var , a, b, [ epsrel, epsabs, limit ] )
```

where the keywords inside the brackets indicate possible optional arguments (entered in any order), such as **epsrel = 1e-6**.

Thus the approximate numerical value of $\int_0^1 e^{x^2}\, dx$ would be the first element of the list returned by **quad_qags (exp (x^2), x, 0, 1)**.

If you call **quad_qags** with an unbound parameter in the integrand, a noun form will be returned which will tell you all the defaults being used.

```
(%i1) quad_qags(a*x,x,0,1);
(%o1) quad_qags(a x, x, 0, 1, epsrel = 1.0E-8, epsabs = 0.0, limit = 200)
```

There is a more efficient Quadpack function available, **quad_qaws**, for integrals which have end point algebraic and/or logarithmic singularites associated with of the form $\int_a^b f(x)\, w(x)\, dx$, if the "weight function" $w(x)$ has the form (note: $f(x)$ is assumed to be well behaved)

$$w(x) = (x-a)^{\alpha}\,(b-x)^{\beta}\ln(x-a)\ln(b-x) \tag{8.1}$$

where $\alpha > -1$ and $\beta > -1$ for convergence. See Sec 8.3.3 for more information on **quad_qaws** and its use.

Example 1

Here is an example of the use of **quad_qags** to compute the numerical value of the integral $\int_0^1 \sqrt{x}\ln(1/x)\,dx$. The integrand $g = x^{1/2}\ln(1/x) = -x^{1/2}\ln(x)$ has the limiting value 0 as $x \rightarrow 0$ from the positive side. Thus the integrand is not singular at $x = 0$, but it is rapidly changing near $x = 0$ so an efficient algorithm is needed. (Our Example 2 will work with an integrand which is singular at $x = 0$.)

Let's check the limit at $x = 0$ and plot the function.

```
(%i2) g:sqrt(x)*log(1/x)$
(%i3) limit(g,x,0,plus);
(%o3)                              0
(%i4) (load(draw),load(qdraw))$
(%i5) qdraw( ex(g,x,0,1) )$
```

Here is that plot.

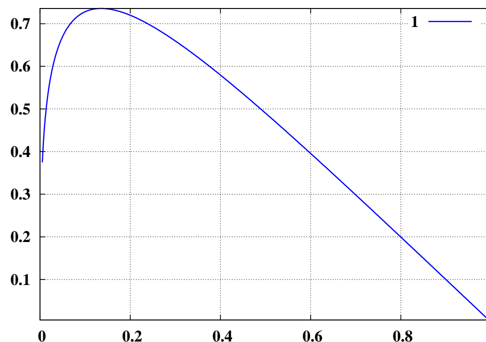

Figure 1: $x^{1/2}\ln(1/x)$

Now we try out **quad_qags** on this integral and compare with the exact answer returned by **integrate**.

```
(%i6) fpprintprec:8$
(%i7) tval:bfloat(integrate(g,x,0,1)),fpprec:20;
(%o7)                          4.4444444b-1
(%i8) qlist:quad_qags(g,x,0,1 );
(%o8)                [0.444444, 4.93432455E-16, 315, 0]
(%i9) abs(first(%) - tval ),fpprec:20;
(%o9)                          8.0182679b-17
```

The first element of the returned list **qlist** is the approximate numerical value of the integral. We have used **integrate** together with **bfloat** and **fpprec** to generate the "true value" good to about **20** digits (see Chapter 9), and we see that the absolute error of the answer returned by **quad_qags** is about 10^{-16}. The second element of **qlist** is the **estimated** absolute error of the returned answer. The third element shows that **315** integrand evaluations were needed to attain the requested (default) relative error **epsrel = 1e-8**. The fourth element is the error code value **0** which indicates no problems were found.

The algorithm used to find an approximation for this integral does not know the "exact" (or true) answer, but does have (at each stage) an estimate of the answer and an estimate of the likely error of this estimated answer (gotten by comparing the new answer with the old answer, for example), and so can compute an estimated relative error (est_abs_err/est_answer) which the code checks against the requested relative error goal supplied by the parameter **epsrel**.

We are assuming that the defaults are being used, and hence **epsabs** has the value **0.0**, so that the convergence criterion used by the code is

$$\text{est_rel_err} \le \text{epsrel} \tag{8.2}$$

or

$$\frac{\text{est_abs_err}}{\text{est_answer}} \le \text{epsrel} \tag{8.3}$$

or

$$\text{est_abs_err} \le \text{epsrel} \times \text{est_answer} \tag{8.4}$$

We can check that the values returned by **quad_qags** are at least consistent with the advertised convergence criterion.

```
(%i10) est_answer : first(qlist);
(%o10)                          0.444444
(%i11) est_abs_err : second(qlist);
(%o11)                       4.93432455E-16
(%i12) est_rel_err : est_abs_err/est_answer;
(%o12)                       1.11022302E-15
```

We see that the **est_rel_err** is much less than **epsrel**.

Example 2

Here we use **quad_qags** with an integrand which is singular at $x = 0$, but the singularity is an "integrable singularity". We want to calculate $\int_0^1 \ln(\sin(x))\,dx$. If you answer enough questions correctly, **integrate** returns an "exact" answer involving the limit of expressions involving the dilogarithm function with a complex argument, and the "answer" is rather long.

We first show that the integrand is singular at $x = 0$, first using Maxima's **limit** function and then using **taylor**. We then use the approximate integrand for small x to show that the indefinite integral is a function of x which has a finite limit at $x = 0$.

```
(%i1) limit(log(sin(x)),x,0,plus);
(%o1)                              minf
(%i2) ix : taylor(log(sin(x)),x,0,2);
                                          2
                                         x
(%o2)/T/                 log(x) + . . . - -- + . . .
                                         6
(%i3) int_ix:integrate(ix,x);
                                         3
                                        x
(%o3)                        x log(x) - -- - x
                                        18
(%i4) limit(int_ix,x,0,plus);
(%o4)                              0
(%i5) assume(eps>0,eps<1)$
(%i6) integrate(ix,x,0,eps);
                                           3
                        18 eps log(eps) - eps  - 18 eps
(%o6)                   -------------------------------
                                     18
```

Output **%o1** indicates that the integrand $\rightarrow -\infty$ as $x \rightarrow 0^{+}$. To see if this is an "integrable singularity", we examine the integrand for x positive but close to 0, using a Taylor series expansion. We let **eps** represent ϵ, a small positive number to be used as the upper limit of a small integration interval.

The approximate integrand, **ix**, shows a logarithmic singularity at $x = 0$. The indefinite integral of the approximate integrand, **int_ix**, is finite as $x \rightarrow 0^{+}$. And finally, the integral of the approximate integrand over $(0, \epsilon)$ is finite. Hence we are dealing with an integrable singularity, and we use **quad_qags** to evaluate the numerical value of the exact integrand over $[0, 1]$.

```
(%i7) quad_qags(log(sin(x)),x,0,1);
(%o7)                [- 1.0567202, 1.1731951E-15, 231, 0]
```

Use of **float(integrate)** with this expression returns a very complicated result involving proposed limits which Maxima has not carried out.

8.2.5 romberg for a Finite Interval

If you have loaded in **romberg**, the syntax **romberg(expr,var, a, b)** returns an estimate of an integral using what is known as "Romberg's Method".

The quadpack function **quad_qags** is more accurate and includes error diagnostics. However we include a short discussion of **romberg** since we will try out the big-float version **bromberg** in Ch. 9.

The default behavior of **romberg** is governed by a relative difference test applied to successive iterations and is determined by the default value of **rombergtol** (which default value is **1e-4**) If you want to change that value, you issue the separate assignment statement: **rombergtol : 1e-15** for example.

```
(%i1) fpprintprec:8$
(%i2) load(romberg);
(%o2) C:/PROGRA~1/MAXIMA~3.1/share/maxima/5.18.1/share/numeric/romberg.lisp
(%i3) [rombergtol,rombergabs,rombergit,rombergmin];
(%o3)                        [1.0E-4, 0.0, 11, 0]
```

We can compare **romberg** with **quad_qags**. Of course the default relative error is different, but by setting **rombergtol** to the value **1e-15** we can then compare their output. We will use the simple integral $\int_0^1 e^x\, dx$:

```
(%i4) tval : bfloat(integrate(exp(x),x,0,1)),fpprec:20;
(%o4)                          1.7182818b0
(%i5) rombergtol:1e-15$
(%i6) romberg(exp(x),x,0,1);
(%o6)                           1.7182818
(%i7) abs(% - tval),fpprec:20;
(%o7)                         3.6660941b-16
(%i8) %/tval,fpprec:20;
(%o8)                         2.1335813b-16
(%i9) quad_qags(exp(x),x,0,1 );
(%o9)                [1.7182818, 1.90767605E-14, 21, 0]
(%i10) abs(%[1] - tval),fpprec:20;
(%o10)                        7.7479797b-17
(%i11) %/tval,fpprec:20;
(%o11)                        4.5091437b-17
```

So we see that **quad_qags** yields an answer with a somewhat smaller relative error than **romberg**.

8.2.6 quad_qags and romberg for Numerical Double Integrals

In Sec.7.6 of Ch.7 we presented the notation (for an exact symbolic double integral):

To evaluate the exact symbolic double integral

$$\int_{u1}^{u2} du \int_{v1(u)}^{v2(u)} dv\, f(u,v) \equiv \int_{u1}^{u2} \left(\int_{v1(u)}^{v2(u)} f(u,v)\, dv \right) du \tag{8.5}$$

we use **integrate** with the syntax:

```
integrate( integrate( f(u,v), v, v1(u), v2(u) ), u, u1, u2 )
```

in which **f(u,v)** can either be an expression depending on the variables **u** and **v**, or a Maxima function, and likewise **v1(u)** and **v2(u)** can either be expressions depending on **u** or Maxima functions.

Both **u** and **v** are "dummy variables", since the value of the resulting double integral does not depend on our choice of symbols for the integration variables; we could just as well use **x** and **y**.

To use **romberg** to find the same double integral (numerically), we use the syntax:

```
romberg( romberg ( f(u,v), v, v1(u), v2(u) ), u, u1, u2 )
```

and to use **quad_qags** we use

```
quad_qags ( quad_qags ( f(u,v), v, v1(u), v2(u) )[1], u, u1, u2 )
```

Example 1

For our first example, we compare **quad_qags** and **romberg** on the double integral:

$$\int_{1}^{3} \left(\int_{0}^{x/2} \frac{x\,y}{x+y}\, dy \right) dx \tag{8.6}$$

comparing both results with the **integrate** result, first using the absolute error, then the relative error.

```
(%i1) fpprintprec:8$
(%i2) g : x*y/(x+y)$
(%i3) tval : bfloat (integrate (integrate (g,y,0,x/2),x,1,3) ),fpprec:20;
Is  x  positive, negative, or zero?

p;
(%o3)                          8.1930239b-1
(%i4) load(romberg)$
(%i5) [rombergtol,rombergabs,rombergit,rombergmin];
(%o5)                     [1.0E-4, 0.0, 11, 0]
(%i6) rombergtol:1e-15$
(%i7) romberg( romberg(g,y,0,x/2),x,1,3);
(%o7)                            0.819302
(%i8) abs(% - tval),fpprec:20;
(%o8)                          1.7298445b-16
(%i9) %/tval,fpprec:20;
(%o9)                          2.1113627b-16
```

```
(%i10) quad_qags ( quad_qags (g,y,0,x/2)[1],x,1,3);
(%o10)                  [0.819302, 9.09608385E-15, 21, 0]
(%i11) abs (%[1] - tval),fpprec:20;
(%o11)                          6.1962154b-17
(%i12) %/tval,fpprec:20;
(%o12)                          7.5627942b-17
```

We see that **quad_qags** is slightly more accurate.

Example 2

For our second example, we consider the double integral

$$\int_0^1 \left(\int_1^{2+x} e^{x-y^2}\, dy \right) dx \tag{8.7}$$

(Note we still have **rombergtol** set to **1e-15**).

```
(%i13) g : exp(x-y^2)$
(%i14) tval : bfloat(integrate( integrate(g,y,1,2+x),x,0,1)),fpprec:20;
(%o14)                          2.3846836b-1
(%i15) romberg( romberg(g,y,1,2+x),x,0,1);
`romberg' failed to converge
                                  2
                              x - y
(%o15)     romberg(romberg(%e      , y, 1.0, x + 2.0), x, 0.0, 1.0)
(%i16) quad_qags( quad_qags(g,y,1,2+x)[1],x,0,1);
(%o16)                  [0.238468, 2.64753066E-15, 21, 0]
(%i17) abs (%[1] - tval),fpprec:20;
(%o17)                          4.229659b-18
(%i18) %/tval,fpprec:20;
(%o18)                          1.7736772b-17
```

We see that **romberg** failed to converge to an answer. If you decrease the value of `rombergtol`, you can eventually get an answer from **romberg** for this double integral.

Maxima coordinator Robert Dodier has (on the Mailing List) commented on the general preference for using **quad_qags** instead of **romberg**:

> ... the quadpack functions are stronger than romberg: they succeed on problems for which romberg fails, they product diagnostic codes when they do fail, they produce the same accuracy with fewer function calls when they succeed, and there are specialized methods for various special cases (infinite domain, etc).
>
> I can't think of a problem for which I'd recommend romberg over quadpack.

Dodier has also warned about the lack of accuracy diagnostics with the inner integral done by **quad_qags**:

> A nested numerical integral like this has a couple of drawbacks, whatever the method. (1) The estimated error in the inner integral isn't taken into account in the error estimate for the outer. (2) Methods specifically devised for multi-dimensional integrals are typically more efficient than repeated 1-d integrals.

8.2.7 quad_qagi for an Infinite or Semi-infinite Interval

See Sec.(8.5) for a decision tree for quadrature over an infinite or semi-infinite region.

The syntax is

```
   quad_qagi ( expr, var, a, b, [epsrel, epsabs, limit] ), where
           (a,b) are the limits of integration.
  Thus  you will have (omitting the optional args):
 quad_qagi (expr, var, minf, b ) with b finite,
 quad_qagi ( expr, var, a, inf ), with a finite, or
 quad_qagi (expr, var, minf, inf ).
```

If at least one of **(a,b)** are not equal to **(minf,inf)**, a noun form will be returned.

The Maxima function **quad_qagi** returns the same type of information (approx-integral, est-abs-error, nfe, error-code) in a list that **quad_qags** returns, and the possible error codes returned have the same meaning.

Here we test the syntax and behavior with a simple integrand, first over $[0, \infty]$:

```
(%i1) fpprintprec:8$
(%i2) tval : bfloat( integrate (exp(-x^2),x,0,inf) ),fpprec:20;
(%o2)                            8.8622692b-1
(%i3) quad_qagi(exp(-x^2),x,0,inf);
(%o3)                  [0.886227, 7.10131839E-9, 135, 0]
(%i4) abs(first(%) - tval),fpprec:20;
(%o4)                            7.2688979b-17
```

and next the same simple integrand over $[-\infty, 0]$:

```
(%i5) quad_qagi(exp(-x^2),x,minf,0);
(%o5)                  [0.886227, 7.10131839E-9, 135, 0]
(%i6) abs(first(%) - tval),fpprec:20;
(%o6)                            7.2688979b-17
```

and, finally over $[-\infty, \infty]$:

```
(%i7) tval : bfloat(2*tval),fpprec:20;
(%o7)                             1.7724538b0
(%i8) quad_qagi(exp(-x^2),x,minf,inf);
(%o8)                 [1.7724539, 1.42026368E-8, 270, 0]
(%i9) abs(first(%) - tval),fpprec:20;
(%o9)                            1.4537795b-16
```

Example 1

Here is another simple example:

```
(%i10) g : exp(-x)*x^(5/100)$
(%i11) tval : bfloat( integrate(g,x,0,inf) ),fpprec:20;
(%o11)                           9.7350426b-1
(%i12) quad_qagi(g,x,0,inf);
(%o12)                 [0.973504, 1.2270015E-9, 315, 0]
```

```
(%i13) abs(first(%) - tval),fpprec:20;
(%o13)                          7.9340695b-15
(%i14) %/tval,fpprec:20;
(%o14)                           8.15001b-15
```

We see that the use of the default mode (accepting the default **epsrel:1d-8** and **epsabs:0**) has resulted in an absolute error which is close to the floating point limit and a relative error of the same order of magnitude (because the value of the integral is of order 1).

Example 2

We evaluate the integral $\int_0^\infty e^{-x} \ln(x)\, dx = -\gamma$, where γ is the Euler-Mascheroni constant, **0.5772156649015329**. Maxima has this constant available as **%gamma**, although you need to use either **float** or **bfloat** to get the numerical value. The Maxima function **integrate** cannot make any progress with this integral.

```
(%i15) g : exp(-x)*log(x)$
(%i16) tval : bfloat( integrate(g,x,0,inf) ),fpprec:20;
The number 0 isn't in the domain of gamma_incomplete
 -- an error.  To debug this try debugmode(true);
(%i17) tval : bfloat(-%gamma),fpprec:20;
(%o17)                        - 5.7721566b-1
(%i18) quad_qagi(g,x,0,inf);
(%o18)               [- 0.577216, 5.11052578E-9, 345, 0]
(%i19) abs(first(%) - tval),fpprec:20;
(%o19)                          2.6595919b-15
(%i20) %/tval,fpprec:20;
(%o20)                        - 4.6076226b-15
```

Example 3

A symmetrical version of a Fourier transform pair is defined by the equations

$$g(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} G(k)\, e^{i k x}\, dk \tag{8.8}$$

$$G(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(x)\, e^{-i k x}\, dx \tag{8.9}$$

An example of such a Fourier transform pair which respects this symmetrical definition is: if $g(x) = a\, e^{-b\, x^2}$, then $G(k) = (a/\sqrt{2\, b})\, e^{-k^2/(4\, b)}$.

```
(%i21) assume(a>0,b>0,k>0)$
(%i22) g :a*exp(-b*x^2)$
(%i23) gft:integrate(exp(-%i*k*x)*g,x,minf,inf)/sqrt(2*%pi);
                                                 2
                                                k
                                              - ---
                                                4 b
                                         a %e
(%o23)                                ---------------
                                      sqrt(2) sqrt(b)
```

To check this relation with **quag_qagi**, let's define **f** to be **g** for the case **a, b** and **k** are all set equal to **1**.

```
(%i24) f : subst([a=1,b=1,k=1],g);
                                            2
                                         - x
(%o24)                                 %e
(%i25) fft : subst([a=1,b=1,k=1],gft);
                                         - 1/4
                                       %e
(%o25)                                 -------
                                       sqrt(2)
(%i26) float(fft);
(%o26)                                0.550695
```

If we try to submit an explicitly complex integrand to **quad_qagi** we get a noun form back, indicating failure. (A similar result occurs with **quad_qags**).

```
(%i27) quad_qagi(f*exp(-%i*x),x,minf,inf);
                             2
                          - x  - %i x
(%o27) quad_qagi(%e            , x, minf, inf, epsrel = 1.0E-8, epsabs = 0.0,
                                                                   limit = 200)
```

8.3 Numerical Integration: Sharper Tools

There are specialised Quadpack routines for particular kinds of one dimensional integrals. See Sec.(8.4) for a "decision tree" for a finite region numerical integral using the Quadpack functions.

8.3.1 quad_qag for a General Oscillatory Integrand

The function **quad_qag** is sometimes of use primarily due to its ability to deal with functions with some general oscillatory behavior.

The "key" feature to watch out for is the required fifth slot argument which the manual calls "key". This slot can have any integral value between 1 and 6 inclusive, with the higher values corresponding to "higer order Gauss-Kronrod" integration rules for more complicated oscillatory behavior.

Use the syntax:

```
quad_qag( expr, var, a, b, key, [epsrel, epsabs, limit] )
```

Thus the approximate numerical value of $\int_0^1 e^{x^2}\,dx$ would be the first element of the list returned by `quad_qag ( exp ( x^2 ), x, 0, 1, 3)` using the "key" value **3**.

Example 1

Since the main feature of **quad_qag** is the ability to select a high order quadrature method for a generally oscillating integrand, we begin by comparing **quad_qag** with **quad_qags** for such a case. We consider the integral $\int_0^1 \cos(50\,x)\sin(3\,x)\,e^{-x}\,dx$.

```
(%i1) fpprintprec:8$
(%i2) f : cos(50*x)*sin(3*x)*exp(-x)$
(%i3) tval : bfloat( integrate (f,x,0,1) ),fpprec:20;
(%o3)                              - 1.9145466b-3
(%i4) (load(draw),load(qdraw))$
(%i5) qdraw( ex(f,x,0,1) )$
```

Here is that plot: and here we make the comparison.

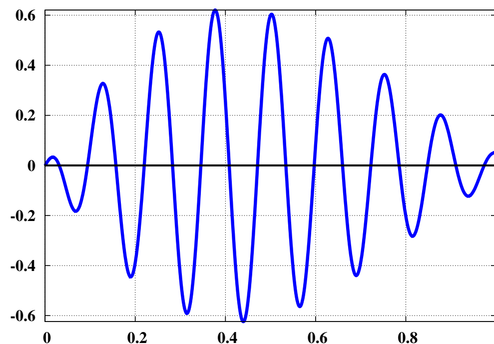

Figure 2: $\cos(50\,x)\sin(3\,x)\,e^{-x}$

```
(%i6) quad_qag(f,x,0,1,6);
(%o6)               [- 0.00191455, 2.86107558E-15, 61, 0]
(%i7) abs(first(%) - tval),fpprec:20;
(%o7)                              9.8573614b-17
(%i8) %/tval,fpprec:20;
(%o8)                              - 5.148666b-14
(%i9) quad_qags(f,x,0,1 );
(%o9)                [- 0.00191455, 2.8536E-15, 315, 0]
(%i10) abs(first(%) - tval),fpprec:20;
(%o10)                             1.1804997b-16
(%i11) %/tval,fpprec:20;
(%o11)                             - 6.1659493b-14
```

We see that both methods returned results with about the same relative error, but that **quad_qag** needed only about one fifth the number of integrand evaluations as compared with **quad_qags**. This extra efficiency of **quad_qag** for this type of integrand may be of interest in certain kinds of intensive numerical work. Naturally, the number of integrand evaluations needed does not necessarily translate simply into time saved, so timing trials would be appropriate when considering this option.

Example 2

We compare **quad_qag** and **quad_qags** with the numerical evaluation of the integral $\int_0^1 e^x\,dx$, which has neither special oscillatory behavior (which **quad_qag** might help with) nor singular behavior (which **quad_qags** might help with).

```
(%i12) tval : bfloat( integrate (exp(x),x,0,1) ),fpprec:20;
(%o12)                           1.7182818b0
(%i13) quad_qag(exp(x),x,0,1,6);
(%o13)               [1.7182818, 1.90767605E-14, 61, 0]
(%i14) abs(first(%) - tval),fpprec:20;
(%o14)                          1.445648b-16
(%i15) %/tval,fpprec:20;
(%o15)                          8.413335b-17
(%i16) quad_qags(exp(x),x,0,1 );
(%o16)               [1.7182818, 1.90767605E-14, 21, 0]
(%i17) abs(first(%) - tval),fpprec:20;
(%o17)                          7.7479797b-17
(%i18) %/tval,fpprec:20;
(%o18)                          4.5091437b-17
```

We see that using **quad_qags** for this function results in slightly smaller relative error while using only two thirds the number of integrand evaluations compared with **quad_qag**.

Example 3

We compare **quad_qag** and **quad_qags** with the numerical evaluation of the integral $\int_0^1 \sqrt{x}\, \ln(x)\, dx$, which has quasi-singular behavior at $x = 0$. See Sec.(8.2.4) for a plot of this integrand.

```
(%i19) f : sqrt(x)*log(1/x)$
(%i20) tval : bfloat( integrate (f,x,0,1) ),fpprec:20;
(%o20)                          4.4444444b-1
(%i21) quad_qag(f,x,0,1,3);
(%o21)                [0.444444, 3.17009685E-9, 961, 0]
(%i22) abs(first(%) - tval),fpprec:20;
(%o22)                          4.7663293b-12
(%i23) %/tval,fpprec:20;
(%o23)                          1.072424b-11
(%i24) quad_qags(f,x,0,1 );
(%o24)               [0.444444, 4.93432455E-16, 315, 0]
(%i25) abs(first(%) - tval),fpprec:20;
(%o25)                          8.0182679b-17
(%i26) %/tval,fpprec:20;
(%o26)                          1.8041102b-16
```

We see that using **quad_qags** for this function (which has a quasi-singular behavior near $x = 0$) returns a much smaller relative error while using only one third the number of integrand evaluations compared with **quad_qag**.

8.3.2 quad_qawo for Fourier Series Coefficients

This function is the most efficient way to find numerical values of Fourier series coefficients, which require finding integrals $\int_a^b f(x) \cos(\omega\, x)\, dx$ or $\int_a^b f(x) \sin(\omega\, x)\, dx$.

This function has the syntax

```
quad_qawo(expr, var , a, b, omega, trig,[epsrel, epsabs, limit, maxp1])
```

For the integral $\int_{-2}^{2}(x+x^4)\cos(3\,x)\,dx$ (default options), use `quad_qawo(x + x^4, x, -2, 2, 3, cos)`.

For the integral $\int_{-2}^{2}(x+x^4)\sin(5\,x)\,dx$ (default options), use `quad_qawo(x + x^4, x, -2, 2, 5, sin)`.
Here we compare **quad_qawo** with **quad_qags** for the integral $\int_{-2}^{2}(x+x^4)\cos(3\,x)\,dx$.

```
(%i1) fpprintprec:8$
(%i2) g : (x+x^4)*cos(3*x)$
(%i3) tval : bfloat( integrate(g,x,-2,2) ),fpprec:20;
(%o3)                           3.6477501b0
(%i4) quad_qawo(x+x^4,x,-2,2,3,cos);
(%o4)                    [3.6477502, 0.0, 25, 0]
(%i5) abs(first(%) - tval),fpprec:20;
(%o5)                          4.0522056b-18
(%i6) %/tval,fpprec:20;
(%o6)                          1.110878b-18
(%i7) quad_qags(g,x,-2,2);
(%o7)              [3.6477502, 8.89609255E-14, 63, 0]
(%i8) abs(first(%) - tval),fpprec:20;
(%o8)                          1.7723046b-15
(%i9) %/tval,fpprec:20;
(%o9)                          4.8586239b-16
```

We see that **quad_qawo** finds the numerical value with "zero" relative error as compared with a "non-zero" relative error using **quad_qags**, and with many less integrand evaluations.

8.3.3 quad_qaws for End Point Algebraic and Logarithmic Singularities

The syntax is:

```
quad_qaws (f(x), x, a, b, alpha, beta, wfun,[epsrel, epsabs, limit])
```

This Maxima function is designed for the efficient evaluation of integrals of the form $\int_a^b f(x)\,w(x)\,dx$ in which the appropriate "singular end point weight function" is chosen from among different versions via the three parameters **wfun**, α (represented by **alpha**), and β (represented by **beta**).

The most general case in which one has both algebraic and logarithmic singularities of the integrand at both end points corresponds to $\alpha \neq 0$ and $\beta \neq 0$ and

$$w(x) = (x-a)^{\alpha}\,(b-x)^{\beta}\,\ln(x-a)\,\ln(b-x) \tag{8.10}$$

The parameters α and β govern the "degree" of algebraic singularity at the end points. One needs both $\alpha > -1$ and $\beta > -1$ for convergence of the integrals.

In particular, one can choose $\alpha = 0$ and/or $\beta = 0$ to handle an algebraic singularity at only one end of the interval or no algebraic singularities at all.

The parameter **wfun** determines the existence and location of possible end point logarithmic singularities of the integrand.

wfun	w(x)
1	$(x-a)^{\alpha}\,(b-x)^{\beta}$
2	$(x-a)^{\alpha}\,(b-x)^{\beta}\ln(x-a)$
3	$(x-a)^{\alpha}\,(b-x)^{\beta}\ln(b-x)$
4	$(x-a)^{\alpha}\,(b-x)^{\beta}\ln(x-a)\,\ln(b-x)$

Example 1: Pure Logarithmic Singularities

For the case that $\alpha = 0$ and $\beta = 0$, there are no end point algebraic singularities, only logarithmic singularities. A simple example is $\int_0^1 \ln(x)\, dx$, which corresponds to $\mathbf{wfun} = 2$:

```
(%i1) fpprintprec:8$
(%i2) tval : bfloat( integrate(log(x),x,0,1) ),fpprec:20;
(%o2)                                   - 1.0b0
(%i3) quad_qaws(1,x,0,1,0,0,2);
(%o3)                     [- 1.0, 9.68809031E-15, 40, 0]
(%i4) abs(first(%) - tval),fpprec:20;
(%o4)                                    0.0b0
(%i5) quad_qags(log(x),x,0,1);
(%o5)                     [- 1.0, 1.11022302E-15, 231, 0]
(%i6) abs(first(%) - tval),fpprec:20;
(%o6)                                    0.0b0
```

which illustrates the efficiency of **quad_qaws** compared to **quad_qags** for this type of integrand.

Example 2: Pure Algebraic Singularity

The case $\mathbf{wfun} = 1$ corresponds to purely algebraic end point singularities.

Here we compare **quad_qaws** with **quad_qags** for the evaluation of the integral $\int_0^1 \frac{\sin(x)}{\sqrt{x}}\, dx$. You will get an exact symbolic answer in terms of **erf(z)** for this integral from **integrate**.

```
(%i15) expand(bfloat(integrate(sin(x)/sqrt(x),x,0,1))),fpprec:20;
(%o15)       1.717976b0 cos(0.25 %pi) - 8.404048b-1 sin(0.25 %pi)
(%i16) tval : bfloat(%),fpprec:20;
(%o16)                               6.205366b-1
(%i17) quad_qaws(sin(x),x,0,1,-1/2,0,1);
(%o17)                [0.620537, 4.31887834E-15, 40, 0]
(%i18) abs(first(%) - tval),fpprec:20;
(%o18)                              8.8091426b-19
(%i19) %/tval,fpprec:20;
(%o19)                              1.4196008b-18
(%i20) quad_qags(sin(x)/sqrt(x),x,0,1);
(%o20)               [0.620537, 3.48387985E-13, 231, 0]
(%i21) abs(first(%) - tval),fpprec:20;
(%o21)                              1.1014138b-16
(%i22) %/tval,fpprec:20;
(%o22)                              1.7749378b-16
```

We see that **quad_qaws** uses about one sixth the number of function evaluations (as compared with **quad_qags**) and returns a much more accurate answer.

Example 3: Both Algebraic and Logarithmic Singularity at an End Point

A simple example is $\int_0^1 \frac{\ln(x)}{\sqrt{x}}\, dx$. The integrand is singular at $x = 0$ but this is an "integrable singularity" since $\sqrt{x}\,\ln(x) \to 0$ as $x \to 0^+$.

```
(%i1) fpprintprec:8$
(%i2) limit(log(x)/sqrt(x),x,0,plus);
(%o2)                                   minf
```

```
(%i3) integrate(log(x)/sqrt(x),x);
(%o3)                  2 (sqrt(x) log(x) - 2 sqrt(x))
(%i4) limit(%,x,0,plus);
(%o4)                                 0
(%i5) tval : bfloat( integrate(log(x)/sqrt(x),x,0,1)),fpprec:20;
(%o5)                              - 4.0b0
(%i6) quad_qaws(1,x,0,1,-1/2,0,2);
(%o6)                  [- 4.0, 3.59396672E-13, 40, 0]
(%i7) abs(first(%) - tval),fpprec:20;
(%o7)                                0.0b0
(%i8) quad_qags(log(x)/sqrt(x),x,0,1);
(%o8)                  [- 4.0, 1.94066985E-13, 315, 0]
(%i9) abs(first(%) - tval),fpprec:20;
(%o9)                            2.6645352b-15
(%i10) %/tval,fpprec:20;
(%o10)                          - 6.6613381b-16
```

Again we see the relative efficiency and accuracy of **quad_qaws** for this type of integral.

8.3.4 quad_qawc for a Cauchy Principal Value Integral

This function has the syntax:

```
quad_qawc (f(x), x, c, a, b,[epsrel, epsabs, limit]).
```

The actual integrand is $g(x) = f(x)/(x - c)$, with dependent variable x, to be integrated over the interval $[a, b]$ and you need to pick out $f(x)$ by hand here. In using **quad_qawc**, the argument c is placed between the name of the variable of integration (here x) and the lower limit of integration.

An integral with a "pole" on the contour does not exist in the strict sense, but if $g(x)$ has a simple pole on the real axis at $x = c$, one defines the Cauchy principal value as the symmetrical limit (with $a < c < b$)

$$P \int_a^b g(x)\, dx = \lim_{\epsilon \to 0^+} \left[\int_a^{c-\epsilon} g(x)\, dx + \int_{c+\epsilon}^b g(x)\, dx \right] \tag{8.11}$$

provided this limit exists. In terms of $f(x)$ this definition becomes

$$P \int_a^b \frac{f(x)}{x - c}\, dx = \lim_{\epsilon \to 0^+} \left[\int_a^{c-\epsilon} \frac{f(x)}{x - c}\, dx + \int_{c+\epsilon}^b \frac{f(x)}{x - c}\, dx \right] \tag{8.12}$$

We can find the default values of the optional method parameters of **quad_qawc** by including an undefined symbol in our call:

```
(%i11) quad_qawc(1/(x^2-1),x,1,0,b);
                   1
(%o11) quad_qawc(------, x, 1, 0, b, epsrel = 1.0E-8, epsabs = 0.0, limit = 200)
                  2
                 x  - 1
```

We see that the default settings cause the algorithm to look at the relative error of succeeding approximations to the numerical answer.

As a simple example of the syntax we consider the principal value integral

$$P \int_0^2 \frac{1}{x^2 - 1}\,dx = P \int_0^2 \frac{1}{(x-1)\,(x+1)}\,dx = -\ln(3)/2 \tag{8.13}$$

We use **assume** to prep **integrate** and then implement the basic definition provided by Eq. (8.11)

```
(%i1) fpprintprec:8$
(%i2) assume(eps>0, eps<1)$
(%i3) integrate(1/(x^2-1),x,0,1-eps) +
               integrate(1/(x^2-1),x,1+eps,2);
                     log(eps + 2)   log(2 - eps)   log(3)
(%o3)                ------------ - ------------ - ------
                          2              2           2
(%i4) limit(%,eps,0,plus);
                                 log(3)
(%o4)                          - ------
                                   2
(%i5) tval : bfloat(%),fpprec:20;
(%o5)                        - 5.4930614b-1
```

We now compare the result returned by **integrate** with the numerical value returned by **quad_qawc**, noting that $f(x) = 1/(1+x)$.

```
(%i6) quad_qawc(1/(1+x),x,1,0,2);
(%o6)                [- 0.549306, 1.51336373E-11, 105, 0]
(%i7) abs(first(%) - tval),fpprec:20;
(%o7)                          6.5665382b-17
(%i8) %/tval,fpprec:20;
(%o8)                        - 1.1954241b-16
```

We see that the relative error of the returned answer is much less than the (default) requested minimum relative error.

If we run **quad_qawc** requesting that convergence be based on absolute error instead of relative error,

```
(%i9) quad_qawc(1/(1+x),x,1,0,2,epsabs=1.0e-10,epsrel=0.0);
(%o9)                [- 0.549306, 1.51336373E-11, 105, 0]
(%i10) abs(first(%) - tval),fpprec:20;
(%o10)                         6.5665382b-17
(%i11) %/tval,fpprec:20;
(%o11)                       - 1.1954241b-16
```

we see no significant difference in the returned accuracy, and again we see that the absolute error of the returned answer is much less than the requested minimum absolute error.

8.3.5 quad_qawf for a Semi-Infinite Range Cosine or Sine Fourier Transform

The function **quad_qawf** calculates a Fourier cosine *or* Fourier sine transform (up to an overall normalization factor) on the semi-infinite interval $[a, \infty]$. If we let **w** stand for the angular frequency in radians, the integrand is **f(x)*cos(w*x)** if the **trig** parameter is **cos**, and the integrand is **f(x)*sin(w*x)** if the **trig** parameter is **sin**.

The calling syntax is

```
quad_qawf (f(x), x, a, w, trig, [epsabs, limit, maxp1, limlst])
```

Thus `quad_qawf (f(x), x, 0, w, 'cos)` will find a numerical approximation to the integral

$$\int_0^{\infty} f(x) \cos(w\, x)\, dx \tag{8.14}$$

If we call **quad_qawf** with undefined parameter(s), we get a look at the default values of the optional method parameters:

```
(%i12) quad_qawf(exp(-a*x),x,0,w,'cos);
                      - a x
(%o12) quad_qawf(%e     , x, 0, w, cos, epsabs = 1.0E-10, limit = 200,
                                                  maxp1 = 100, limlst = 10)
```

The manual has the optional parameter information:

```
The keyword arguments are optional and may be specified in any order.
They all take the form keyword = val. The keyword arguments are:
1. epsabs, the desired absolute error of approximation. Default is 1d-10.
2. limit, the size of the internal work array.
   (limit - limlst)/2 is the maximum number of
   subintervals to use. The default value of limit is 200.
3. maxp1, the maximum number of Chebyshev moments.
      Must be greater than 0. Default is 100.
4. limlst, upper bound on the number of cycles.
    Must be greater than or equal to 3. Default is 10.
```

The manual does not define the meaning of "cycles". There is no "epsrel" parameter used for this function.

Here is the manual example, organised in our way. In this example $w = 1$ and $a = 0$.

```
(%i1) fpprintprec:8$
(%i2) integrate (exp(-x^2)*cos(x), x, 0, inf);
                                 - 1/4
                               %e      sqrt(%pi)
(%o2)                          -----------------
                                       2
(%i3) tval : bfloat(%),fpprec:20;
(%o3)                            6.9019422b-1
(%i4) quad_qawf (exp(-x^2), x, 0, 1, 'cos );
(%o4)                 [0.690194, 2.84846299E-11, 215, 0]
(%i5) abs(first(%) - tval),fpprec:20;
(%o5)                            3.909904b-17
(%i6) %/tval,fpprec:20;
(%o6)                            5.664933b-17
```

We see that the absolute error of the returned answer is much less than the (default) requested minimum absolute error.

8.4 Finite Region of Integration Decision Tree

If you are in a hurry, use **quad_qags**. [*]

If your definite integral is over a finite region of integration **[a, b]**, then

1. If the integrand has the form **w(x)*f(x)**, where **f(x)** is a smooth function over the region of integration, then

   - If **w(x)** has the form of either **cos(c*x)** or **sin(c*x)**, where **c** is a constant, use **quad_qawo**.
   - Else if the factor **w(x)** has the form (note the limits of integration are **[a, b]**

     ```
     (x - a)^ae * (b - x)^be * (log(x - a))^na * (log(b - x))^nb
     ```

     where **na**, **nb** have the values **0** or **1**, and both **ae** and **be** are greater than -1, then use **quad_qaws**. (This is a case where we need a routine which is designed to handle end point singularities.)
   - Else if the factor **w(x)** is **1/(x - c)** for some constant **c** with $a < c < b$, then use **quad_qawc**, the Cauchy principle value routine.

2. Otherwise, if you do not care too much about possible inefficient use of computer time, and do not want to further analyze the problem, use **quad_qags**.

3. Otherwise, if the integrand is **smooth**, use **quad_qag**.

4. Otherwise, if there are **discontinuities or singularities** of the integrand or of the derivative of the integrand, and you know where they are, split the integration range at these points and separately integrate over each subinterval.

5. Otherwise, if the integrand has **end point singularities**, use **quad_qags**.

6. Otherwise, if the integrand has an oscillatory behavior of nonspecific type, and no singularities, use **quad_qag** with the fifth **key** slot containing the value **6**.

7. Otherwise, use **quad_qags**.

[*] These "decision trees" are adapted from the web page
http://people.scs.fsu.edu/~burkardt/f77_src/quadpack/quadpack.html.

8.5 Semi-infinite or Infinite Region of Integration Decision Tree

Here is the tree for the semi-infinite or infinite domain case:

```
A.  If the integration domain is semi-infinite or from minus to plus infinity,
       consider first making a suitable change of variables to obtain a
           finite domain and using the finite interval decision tree described
           above.

B.  Otherwise, if the integrand decays rapidly to zero, truncate the
      integration interval and use the finite interval decision tree.
C.  Otherwise, if the integrand oscillates over the entire infinite range,

       1. If integral is a Fourier transform, use quad_qawf.

       2. else if the integral is not a Fourier transform, then sum the
           successive positive and negative contributions by integrating
           between the zeroes of the integrand, using the finite
           interval criteria.

D.  Otherwise, if you are not constrained by computer time, and do not
     wish to analyze the problem further, use quad_qagi.

E.  Otherwise, if the integrand has a non-smooth behavior in the range of
     integration, and you know where it occurs, split off these regions
     and use the appropriate finite range routines to integrate over them.
     Then begin this semi-infinite or infinite case tree again to handle
     the remainder of the region.

F.  Otherwise, truncation of the interval, or application of a suitable
     transformation for reducing the problem to a finite range may be
     possible.

G.  Otherwise use quad_qagi.
```

Maxima by Example:
Ch.9: Bigfloats and Arbitrary Precision Quadrature *

Edwin L. Woollett

September 16, 2010

Contents

9.1 Introduction 4
9.2 The Use of Bigfloat Numbers in Maxima 4
- 9.2.1 Bigfloat Numbers Using **bfloat**, **fpprec**, and **fpprintprec** 4
- 9.2.2 Using **print** and **printf** with Bigfloats 8
- 9.2.3 Adding Bigfloats Having Differing Precision 11
- 9.2.4 Polynomial Roots Using **bfallroots** 12
- 9.2.5 Bigfloat Number Gaps and Binary Arithmetic 14
- 9.2.6 Effect of Floating Point Precision on Function Evaluation 15

9.3 Arbitrary Precision Quadrature with Maxima 16
- 9.3.1 Using **bromberg** for Arbitrary Precision Quadrature 16
- 9.3.2 A **Double Exponential** Quadrature Method for $\mathbf{a} \le \mathbf{x} < \infty$ 19
- 9.3.3 The **tanh-sinh** Quadrature Method for $\mathbf{a} \le \mathbf{x} \le \mathbf{b}$ 22
- 9.3.4 The **Gauss-Legendre** Quadrature Method for $\mathbf{a} \le \mathbf{x} \le \mathbf{b}$ 28

*This version uses **Maxima 5.18.1**. This is a live document. Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions to `woollett@charter.net`

Preface

```
COPYING AND DISTRIBUTION POLICY
This document is part of a series of notes titled
"Maxima by Example" and is made available
via the author's webpage http://www.csulb.edu/~woollett/
to aid new users of the Maxima computer algebra system.

NON-PROFIT PRINTING AND DISTRIBUTION IS PERMITTED.

You may make copies of this document and distribute them
to others as long as you charge no more than the costs of printing.

These notes (with some modifications) will be published in book form
eventually via Lulu.com in an arrangement which will continue
to allow unlimited free download of the pdf files as well as the option
of ordering a low cost paperbound version of these notes.
```

Feedback from readers is the best way for this series of notes to become more helpful to new users of Maxima. *All* comments and suggestions for improvements will be appreciated and carefully considered.

```
LOADING FILES
The defaults allow you to use the brief version load(brmbrg) to load in the
Maxima file brmbrg.lisp.
To load in your own file, such as qbromberg.mac (used in this chapter),
using the brief version load(qbromberg), you either need to place
qbromberg.mac in one of the folders Maxima searches by default, or
else put a line like:

file_search_maxima : append(["c:/work3/###.{mac,mc}"],file_search_maxima )$

in your personal startup file maxima-init.mac (see Ch. 1, Introduction to Maxima
for more information about this).

Otherwise you need to provide a complete path in double quotes,
as in load("c:/work3/qbromberg.mac"),

We always use the brief load version in our examples, which are generated
using the Xmaxima graphics interface on a Windows XP computer, and copied
into a fancy verbatim environment in a latex file which uses the fancyvrb
and color packages.

We use qdraw.mac for plots (see Ch.5), which uses draw2d defined
in share/draw/draw.lisp.
```

```
  Maxima, a Computer Algebra System.
  Some numerical results depend on the Lisp version used.
  This chapter uses Version 5.18.1 (2009) using Lisp GNU
  Common Lisp (GCL) GCL 2.6.8 (aka GCL).
  http://maxima.sourceforge.net/
```

The author would like to thank the Maxima developers for their friendly help via the Maxima mailing list.

9.1 Introduction

This chapter is divided into two sections.

In the first section we discuss the use of **bfloat**, including examples which also involve **fpprec**, **bfloatp**, **bfallroots**, **fpprintprec**, **print**, and **printf**. The second section of Chapter 9 presents examples of the use of Maxima for arbitrary precision quadrature (numerical integration). (In Chapter 8, we gave numerous examples of numerical integration using the Quadpack functions such as **quad_qags** as well as **romberg**. Those examples all accepted the default floating point precision of Maxima).

Chapter 10 covers both Fourier transform and Laplace transform type integrals. Chapter 11 presents tools for the use of fast Fourier transforms with examples of use.

Software files developed for Ch. 9 and available on the author's web page include:
1. fdf.mac, 2. qbromberg.mac. ,
3. quad_de.mac, 4. quad_ts.mac, 5. quad_gs.mac.

9.2 The Use of Bigfloat Numbers in Maxima

9.2.1 Bigfloat Numbers Using bfloat, fpprec, and fpprintprec

bfloat(expr) converts all numbers and functions of numbers in **expr** to bigfloat numbers. You can enter explicit bigfloat numbers using the notation **2.38b0**, or **2.38b7**, or **2.38b-4**, for example. **bfloatp(val)** returns **true** if **val** is a bigfloat number, otherwise **false** is returned.

The number of significant digits in the resulting bigfloat is specified by the parameter **fpprec**, whose default value is **16**. The setting of **fpprec** does not affect computations on ordinary floating point numbers.

The underlying Lisp code has two variables: 1. **$fpprec**, which defines the Maxima variable **fpprec** (which determines the number of DECIMAL digits requested for arithmetic), and 2. **fpprec** which is related to the number of bits used for the fractional part of the bigfloat, and which can be accessed from Maxima using **?fpprec**. We can also use the **:lisp foobar** construct to look at a lisp variable **foobar** from "inside" the Lisp interpreter. You do not end with a semi-colon; you just press ENTER to get the response, and the input line number of the Maxima prompt does not advance.

```
(%i1) fpprec;
(%o1)                               16
(%i2) ?fpprec;
(%o2)                               56
(%i3) :lisp $fpprec
16
(%i3) :lisp fpprec
56
```

We discuss some effects of the fact that computer arithmetic is carried out with binary bit representation of decimal numbers in Sec. 9.2.5.

Controlling Printed Digits with fpprintprec

When using bigfloat numbers, the screen can quickly fill up with numbers with many digits, and **fpprintprec** allows you to control how many digits are displayed. For bigfloat numbers, when **fpprintprec** has a value between **2** and **fpprec** (inclusive), the number of digits printed is equal to **fpprintprec**. Otherwise, **fpprintprec** can be **0**, or greater than **fpprec**, in which case the number of digits printed is equal to **fpprec**. **fpprintprec** cannot be **1**. The setting of **fpprintprec** does not affect the precision of the bigfloat arithmetic carried out, only the setting of **fpprec** matters.

The parameter **fpprec** can be used as a local variable in a function defined using **block**, and set to a local value which does not affect the global setting. Such a locally defined value of **fpprec** governs the bigfloat calculations in that function and in any functions called by that function, and in any third layer functions called by the secondary layer functions, etc. **fpprintprec** can be set to a value inside a function defined with **block**,without changing the global value, provided you include the name **fpprintprec** in the local variables bracket **[ ]**:

```
(%i1) [fpprec,fpprintprec];
(%o1)                               [16, 0]
(%i2) piby2 : block([fpprintprec,fpprec:30,val],
                 val:bfloat(%pi/2),
                 fpprintprec:8,
                 disp(val),
                  print("  ",val),
                  val);
                                  1.5707963b0

   1.5707963b0
(%o2)                  1.57079632679489661923132169164b0
(%i3) [fpprec,fpprintprec];
(%o3)                               [16, 0]
```

For simple **bfloat** uses in interactive mode, one can use the syntax **bfloat-job, fpprec:fp ;** which implicitly uses the **ev(...)** construct with temporary settings of global flags as in

```
(%i1) bfloat(%pi),fpprec:20;
(%o1)                       3.1415926535897932385b0
(%i2) slength(string(%));
(%o2)                                 23
(%i3) fpprec;
(%o3)                                 16
(%i4) bfloat(%pi),fpprec:40;
(%o4)             3.141592653589793238462643383279502884197b0
(%i5) slength(string(%));
(%o5)                                 43
(%i6) fpprec;
(%o6)                                 16
(%i7) tval : bfloat(integrate(exp(x),x,-1,1)),fpprec:30;
(%o7)                  2.35040238728760291376476370119b0
(%i8) slength(string(%));
(%o8)                                 33
```

Next we illustrate passing both a bigfloat number as well as local values of **fpprec** and **fpprintprec** to a second function. Function **f1** is designed to call function **f2**:

```
(%i1) f2(w) := block([v2 ],
            disp(" in f2 "),
            display([w,fpprec,fpprintprec]),
             v2 : sin(w),
             display(v2),
             print("  "),
              v2 )$
(%i2) f1(x,fp,fprt) :=
           block([fpprintprec,fpprec:fp,v1],
             fpprintprec:fprt,
             disp(" in f1 "),
              display([x,fpprec,fpprintprec]),
              v1 : f2(bfloat(x))^2,
              print(" in f1, v1 = ",v1),
              v1 )$
```

And here we call **f1** with values **x = 0.5, fp = 30, fprt = 8**:

```
(%i3) f1(0.5,30,8);
                               in f1
                    [x, fpprec, fpprintprec] = [0.5, 30, 8]
                               in f2
                  [w, fpprec, fpprintprec] = [5.0b-1, 30, 8]
                             v2 = 4.7942553b-1
 in f1, v1 =  2.2984884b-1
(%o3)                2.29848847065930141299531696279b-1
(%i4) [fpprec,fpprintprec];
(%o4)                             [16, 0]
```

We see that the called function (**f2**) maintains the values of **fpprec** and **fpprintprec** which exist in the calling function (**f1**).

Bigfloat numbers are "contagious" in the sense that, for example, multiplying (or adding) an integer or ordinary float number with a bigfloat results in a bigfloat number. In the above example **sin(w)** is a bigfloat since **w** is one.

```
(%i5) 1 + 2.0b0;
(%o5)                              3.0b0
(%i6) 1.0 + 2.0b0;
(%o6)                              3.0b0
(%i7) sin(0.5b0);
(%o7)                      4.79425538604203b-1
```

The Maxima symbol **%pi** is not automatically converted by contagion (in the present version of Maxima), and an extra use of **bfloat** does the conversion.

```
(%i8) a:2.38b-1$
(%i9) exp(%pi*a);
                                  2.38b-1 %pi
(%o9)                           %e
(%i10) bfloat(%);
(%o10)                     2.112134508503361b0
```

A recent comment by Maxima developer Volker van Nek on the Maxima mailing list illustrates how the setting of **fpprec** can affect what is printed to the screen when doing bigfloat arithmetic.

```
(%i11) 1b0 + 1b-25,fpprec:26;
(%o11)                  1.0000000000000000000000001b0
(%i12) slength(string(%));
(%o12)                               29
```

Note that **1b-25** is the bigfloat version of 10^{-25}. In the case above the value of **fpprec** is large enough to see the effect of the bigfloat addition. The number of digits identified by the conversion of the number to a string and then finding the length of the string is $29 - 3 = 26$ since the three string characters $[., \mathrm{b}, 0]$ do not contribute to the precision of the number. If we leave the **fpprec** at **26** and add to **1** the bigfloat equivalent of 10^{-26}, Maxima prints out a bigfloat version of **1**, but if we subtract from that answer $1\mathrm{b}0$ we do not get zero:

```
(%i13) 1b0 + 1b-25,fpprec:26;
(%o13)                  1.0000000000000000000000001b0
(%i14) slength(string(%));
(%o14)                               29
(%i15) 1b0 +10b0^(-26),fpprec:26;
(%o15)                              1.0b0
(%i16) % - 1b0,fpprec:26;
(%o16)                 9.6935228033557930648993206b-27
(%i17) slength(string(%));
(%o17)                               31
```

Of the thirty one characters present in this string, the five characters $[., \mathrm{b}, -, 2, 7]$ do not contribute to the precision, , leaving **26** digits of precision. Note that the setting of **fpprintprec** was **0** in the above, which provides for printing of the same number of significant digits as the value of **fpprec**.

Here are three ways to enter the bigfloat equivalent of $1/10 = 10^{-1} = 0.1$:

```
(%i18) [1b-1,bfloat(10^(-1)),10b0^(-1)];
(%o18)                    [1.0b-1, 1.0b-1, 1.0b-1]
```

We next show that once bigfloat numbers are in play inside a **block** function in which a local value of **fpprec** is set, the subsequent arithmetic is the same whether or not you wrap each operation with **bfloat**.

```
(%i19) test1(h,j,fp):=
  block([fpprec:fp,u,a,x,w],
   u:bfloat(h*j),
   a:bfloat(exp(-u)),
   x:bfloat(exp(u-a)),
   w:bfloat(exp(-a)+x),
   [x,w])$
(%i20) test2(h,j,fp):=
  block([fpprec:fp,u,a,x,w],
   u:bfloat(h*j),
   a:exp(-u),
   x:exp(u-a),
   w:exp(-a)+x,
   [x,w])$
(%i21) [xt,wt]:test1(1/16,9,60)$
(%i22) [x1,w1]:test1(1/16,9,40)$
(%i23) [x2,w2]:test2(1/16,9,40)$
```

```
(%i24) [x1,w1]-[xt,wt],fpprec:60;
(%o24) [- 2.29699398191727953386796229322249912456243487302509210328785b-42,
          - 9.07907621847706474924315736438559811433348894389391230442376b-42]
(%i25) [x2,w2]-[xt,wt],fpprec:60;
(%o25) [- 2.29699398191727953386796229322249912456243487302509210328785b-42,
          - 9.07907621847706474924315736438559811433348894389391230442376b-42]
(%i26) bfloat([x1,w1]-[xt,wt]),fpprec:60;
(%o26) [- 2.29699398191727953386796229322249912456243487302509210328785b-42,
          - 9.07907621847706474924315736438559811433348894389391230442376b-42]
(%i27) map('bfloatp,%o5);
(%o27)                         [true, true]
(%i28) map('bfloatp,%o4);
(%o28)                         [true, true]
```

In the above, we used the "completely **bfloat** wrapped" version test1(..) to define answers with **60** digit precision, and then used test1(..) and test2(..) to compute comparison answers at **40** digit precision. We see that there is no difference in precision between the answers returned by the two test versions (each using 40 digit precision).

We also see, from the output **%08**, that **arith_job, fpprec:60;** using interactive mode produces the same answer (with bigfloats already in play) whether or not the "**arithmetic_job**" is wrapped in **bfloat**. The numbers returned by both test versions are bigfloats, as indicated by the last two outputs.

9.2.2 Using print and printf with Bigfloats

In Sec. 9.2.1 we described the relations between the settings of **fpprec** and **fpprintprec**. Once you have generated a bigfloat with some precision, it is convenient to be able to control how many digits are displayed. We start with the use of **print**. If you start with the global default value of **16** for **fpprec** and the default value of **0** for **fpprintprec**, you can use a simple one line command for a low number of digits, as shown in the following. We first define a bigfloat **bf1** to have **fpprec = 45** digits of precision:

```
(%i1) [fpprec,fpprintprec];
(%o1)                              [16, 0]
(%i2) bf1:bfloat(integrate(exp(x),x,-1,1)),fpprec:45;
(%o2)          2.35040238728760291376476370119120163031143596b0
(%i3) slength(string(%));
(%o3)                                48
```

We then use **print** with **fpprintprec** to get increasing numbers of digits on the screen:

```
(%i4) print(bf1),fpprintprec:12$
2.35040238728b0
(%i5) [fpprec,fpprintprec];
(%o5)                              [16, 0]
(%i6) print(bf1),fpprintprec:15$
2.3504023872876b0
(%i7) [fpprec,fpprintprec];
(%o7)                              [16, 0]
(%i8) print(bf1),fpprintprec:16$
2.35040238728760291376476370119120163031143596b0
(%i9) [fpprec,fpprintprec];
(%o9)                              [16, 0]
(%i10) slength(string(%o8));
(%o10)                               48
```

As you see above, when **fpprintprec** reaches the global value of **fpprec = 16** all **45** digits are printed. To control the number of printed digits, you need to locally set the value of **fpprec** as shown here:

```
(%i11) print(bf1),fpprec:20,fpprintprec:18$
2.35040238728760291b0
```

To use this construct in a **do** loop, wrap it in **ev(...)**:

```
(%i12) for j:14 thru 18 do ev(print(bf1),fpprec:j+2,fpprintprec:j)$
2.3504023872876b0
2.3504023872876b0
2.350402387287602b0
2.3504023872876029b0
2.35040238728760291b0
```

A more formal approach is to define a small function which we call **bfprint**:

```
(%i13) bfprint(bf,fpp):=
  block([fpprec, fpprintprec ],
        fpprec : fpp+2,
        fpprintprec:fpp,
        print("   number of digits = ",fpp),
        print("  ",bf) )$
```

with the behavior:

```
(%i14) bfprint(bf1,24)$
   number of digits =  24
   2.35040238728760291376476b0
```

Using printf with bigfloats

We first show some interactive use of **printf** with bigfloats.

```
(%i1) bf:bfloat(exp(-20)),fpprec:30;
(%o1)                2.06115362243855782796594038016b-9
(%i2) slength(string(%));
(%o2)                                 34
(%i3) printf(true,"~d~a",3,string(bf))$
32.06115362243855782796594038016b-9
```

The format string is enclosed in double quotes, with **~d** used for an integer, **~f** used for a floating point number, **~a** used for a Maxima string, **~e** used for exponential display of a floating point number, and **~h** used for a bigfloat number. You can include the newline instruction with **~%** anywhere and as many times as you wish. In the example above, we used the string formatting to display the bigfloat number bf, which required that bf be converting to a Maxima string using **string**. Because we did not include any spaces between the integer format instruction **~d** and the string format character **~a**, we get **32.0...** instead of **3   2.0...**.

```
(%i4) printf(true,"  ~d~a",3,string(bf))$
  32.06115362243855782796594038016b-9
(%i5) printf(true,"  ~d    ~a",3,string(bf))$
  3    2.06115362243855782796594038016b-9
```

```
(%i6) (printf(true,"  ~d    ~a",3,string(bf)),
           printf(true,"  ~d    ~a",3,string(bf)))$
  3    2.061153622438557827965940380 16b-9  3    2.061153622438557827965940380 16b-9
(%i7) (printf(true,"  ~d    ~a~%",3,string(bf)),
           printf(true,"  ~d    ~a",3,string(bf)))$
  3    2.061153622438557827965940380 16b-9
  3    2.061153622438557827965940380 16b-9
```

To get the output on successive lines we had to include the newline instruction ~%. To control the number of significant figures displayed, we use **fpprintprec**:

```
(%i8) fpprintprec:8$
(%i9) printf(true,"  ~d    ~a",3,string(bf))$
  3    2.0611536b-9
```

Next let's show what we get if we use the other options:

```
(%i10) printf(true,"  ~d    ~f",3,bf)$
  3    0.0000000020611536224385579
(%i11) printf(true,"  ~d    ~e",3,bf)$
  3    2.0611536224385579E-9
(%i12) printf(true,"  ~d    ~h",3,bf)$
  3    0.0000000020611536
```

A Table of Bigfloats using block and printf

Here is an example of using **printf** with bigfloats inside a **block** to make a table.

```
(%i1) print_test(fp) :=
        block([fpprec,fpprintprec,val],
         fpprec : fp,
         fpprintprec : 8,
         display(fpprec),
         print("  k          value "),
         print("  "),
         for k thru 4 do
          ( val : bfloat(exp(k^2)),
            printf(true,"  ~d       ~a  ~%",k,string(val) ) ))$
(%i2) print_test(30)$
                                 fpprec = 30

  k          value

  1       2.7182818b0
  2       5.459815b1
  3       8.1030839b3
  4       8.8861105b6
```

Note the crucial use of the newline instruction ~% to get the table output. Some general use examples of **printf** can be found in the Maxima manual and in the file

```
C:\Program Files\Maxima-5.17.1\share\maxima\5.17.1\share\
                              contrib\stringproc\rtestprintf.mac
```

We can use **printf** for the titles and empty lines with the alternative version. We first define an alternative function **print_test2**:

```
(%i3) print_test2(fp) :=
       block([fpprec,fpprintprec,val],
        fpprec : fp,
        fpprintprec : 8,
        display(fpprec),
        printf(true,"~%   ~a           ~a ~%~%",k,value),
        for k thru 4 do
         ( val : bfloat(exp(k^2)),
           printf(true,"  ~d       ~a  ~%",k,string(val) ) ))$
```

Here we try out the alternative function with **fp = 30**:

```
(%i4) print_test2(30)$
                                    fpprec = 30

  k          value

  1       2.7182818b0
  2       5.459815b1
  3       8.1030839b3
  4       8.8861105b6
```

9.2.3 Adding Bigfloats Having Differing Precision

If **A** and **B** are bigfloats with different precisions, the precision of the sum **(A + B)** is the precision of the least precise number. As an example, we calculate an approximation to π using both 30 and 50 digit precision, and add the numbers using 40 digit precision, and then using 60 digit precision. In both cases, the result has 30 digit precision.

```
(%i1) fpprintprec:8$
(%i2) pi50 : bfloat(%pi),fpprec:50;
(%o2)                           3.1415926b0
(%i3) pi30 : bfloat(%pi),fpprec:30;
(%o3)                           3.1415926b0
(%i4) abs(pi30 - pi50),fpprec:60;
(%o4)                          1.6956855b-31
(%i5) twopi : bfloat(2*%pi),fpprec:60;
(%o5)                           6.2831853b0
(%i6) pisum40 : pi30 + pi50,fpprec:40;
(%o6)                           6.2831853b0
(%i7) abs(pisum40 - twopi),fpprec:60;
(%o7)                          1.6956855b-31
(%i8) pisum60 : pi30 + pi50,fpprec:60;
(%o8)                           6.2831853b0
(%i9) abs(pisum60 - twopi),fpprec:60;
(%o9)                          1.6956855b-31
```

9.2.4 Polynomial Roots Using bfallroots

The Maxima function **bfallroots** has the same syntax as **allroots**, and computes numerical approximations of the real and complex roots of a polynomial or polynomial equation of one variable. In all respects, **bfallroots** is identical to **allroots** except that **bfallroots** computes the roots using bigfloats. The source code of **bfallroots** with some comments is in the file **cpoly.lisp** in the **src** directory.

Our example is a cubic equation

$$x^3 + 3\,x^2 - 13\,x - 38 = 0 \tag{9.1}$$

```
(%i1) (ratprint:false, fpprintprec:8)$
(%i2) e : x^3+3*x^2-13*x-38$
```

Let **r** be the list of symbolic roots returned by **solve**. Then **first(r)** will be the first symbolic root. For brevity, we only show this first symbolic root.

```
(%i3) r : map('rhs, solve(e))$
(%i4) first(r);
                              - 3/2
          sqrt(3) %i    1    3       sqrt(2101) %i    23 1/3
(%o4) (- ---------- - -) (-------------------- + --)
                 2         2             2              2
                                                           sqrt(3) %i    1
                                                       16 (---------- - -)
                                                               2         2
                                                 + -------------------------------- - 1
                                                           - 3/2
                                                          3       sqrt(2101) %i    23 1/3
                                                       3 (-------------------- + --)
                                                                   2              2
```

Now let **rf16** be the default floating point numerical value of the symbolic roots returned by **solve**, in which we use **rectform(float(r))**.

```
(%i5) rf16 : rectform(float(r));
(%o5)      [- 4.4408921E-16 %i - 2.8051181, - 3.7793103, 3.5844283]
```

We see that due to floating point arithmetic errors, the first root in the returned list has a very small imaginary part. We will find that there are only real roots by using **allroots**:

```
(%i6) rar16 : map('rhs,allroots(e));
(%o6)                  [3.5844283, - 2.8051181, - 3.7793103]
```

Now let's compare with **bfallroots** using the default **fpprec = 16**.

```
(%i7) rbf16 : map('rhs,bfallroots(e));
(%o7)              [3.5844283b0, - 2.805118b0, - 3.7793102b0]
(%i8) rbf16 - rar16;
(%o8)                [0.0b0, 1.3322676b-15, - 1.2767564b-15]
```

We see small differences which are of the order of the default **16** digit precision being used.

Next, we use **bfallroots** with **fpprec = 30** and compare the results.

```
(%i9) rbf30 : map('rhs,bfallroots(e)),fpprec:30;
(%o9)            [3.5844283b0, - 2.805118b0, - 3.7793102b0]
(%i10) rbf30 - rbf16,fpprec:30;
(%o10)          [9.2452682b-18, - 7.0517343b-17, 5.7609241b-18]
```

Instead of using bigfloats via **bfallroots**, we can use bigfloats by using **bfloat** directly on the exact symbolic roots **r** returned by **solve**.

```
(%i11) rrbf30 : rectform(bfloat(r)),fpprec:30;
(%o11) [- 7.888609b-31 %i - 2.805118b0, 1.1832913b-30 %i - 3.7793102b0,
                                               9.8607613b-32 %i + 3.5844283b0]
(%i12) rrbf30 : realpart(rrbf30);
(%o12)            [- 2.805118b0, - 3.7793102b0, 3.5844283b0]
(%i13) first(rrbf30) - second(rbf30),fpprec:30;
(%o13)                          1.5777218b-30
(%i14) second(rrbf30) - third(rbf30),fpprec:30;
(%o14)                         - 3.1554436b-30
(%i15) third(rrbf30) - first(rbf30),fpprec:30;
(%o15)                              0.0b0
```

We see that the differences are of the order of the **30** digit precision being used.

The conventional wisdom on the Maxima mailing list is that more accurate results are found if the expression **e** is multiplied by **%i**, as in **bfallroots(%i*e)**. We will test that advice here by using **%i*e** with **30** digit precision, and also with **40** digit precision (in order to have a "true value" to use for comparisons).

```
(%i16) rbf30i : map('rhs,bfallroots(%i*e)),fpprec:30;
(%o16) [3.5844283b0 - 1.3363823b-51 %i, 1.0114221b-50 %i - 2.805118b0,
                                             - 8.7778395b-51 %i - 3.7793102b0]
(%i17) rbf30i : realpart(rbf30i);
(%o17)            [3.5844283b0, - 2.805118b0, - 3.7793102b0]
(%i18) rbf30i - rbf30,fpprec:30;
(%o18)              [0.0b0, 7.888609b-31, - 1.5777218b-30]
(%i19) rbf40i : map('rhs,bfallroots(%i*e)),fpprec:40;
(%o19) [3.5844283b0 - 2.687686b-55 %i, 2.0315714b-54 %i - 2.805118b0,
                                             - 1.7628028b-54 %i - 3.7793102b0]
(%i20) rbf40i : realpart(rbf40i);
(%o20)            [3.5844283b0, - 2.805118b0, - 3.7793102b0]
(%i21) rbf30 - rbf40i,fpprec:40;
(%o21)         [- 2.2105568b-32, - 1.7332203b-30, 2.5441868b-30]
(%i22) rbf30i - rbf40i,fpprec:40;
(%o22)         [- 2.2105568b-32, - 9.4435949b-31, 9.6646505b-31]
```

We see that two out of three of the **30** digit precision roots found using the **%i*e** method are slightly more accurate.

9.2.5 Bigfloat Number Gaps and Binary Arithmetic

fpprec as set by the user is the number of DECIMAL digits being requested. In fact, the actual arithmetic is carried out with binary arithmetic. Due to the inevitably finite number of binary bits used to represent a floating point number there will be a range of floating point numbers which are not recognised as different.

For a simple example, let's take the case **fpprec = 4**. Consider the gap around the number `x:bfloat(2/3)` whose magnitude is less than **1**. We will find that **?fpprec** has the value **16** and that Maxima behaves as if (for this case) the fractional part of a bigfloat number is represented by the state of a system consisting of **18** binary bits.

Let $\mathbf{u} = \mathbf{2^{-18}}$. If we let $\mathbf{x1} = \mathbf{x} + \mathbf{u}$ we get a number which is treated as having a nonzero difference from $\mathbf{x}$. However, if we let $\mathbf{w}$ be a number which is one significant digit less than $\mathbf{u}$, and define $\mathbf{x2} = \mathbf{x} + \mathbf{w}$, $\mathbf{x2}$ is treated as having **zero** difference from $\mathbf{x}$. Thus the gap in bigfloats around our chosen $\mathbf{x}$ is $\mathbf{ulp} = \mathbf{2} \cdot \mathbf{2^{-18}} = \mathbf{2^{-17}}$, and this gap should be the same size (as long as **fpprec = 4**) for any bigfloat with a magnitude less than **1**.

If we consider a bigfloat number whose decimal magnitude is less than **1**, its value is represented by a "fractional binary number". For the case that this fractional binary number is the state of **18** (binary) bits, the smallest base 2 number which can occur is the state in which all bits are off (0) except the least significant bit which is on (1), and the decimal equivalent of this fractional binary number is precisely $\mathbf{2^{-18}}$. Adding two bigfloats (each of which has a decimal magnitude less than **1**) when each is represented by the state of an **18** binary bit system (interpreted as a fractional binary number), it is not possible to increase the value of any one bigfloat by less than this smallest base **2** number.

```
(%i1) fpprec:4$
(%i2) ?fpprec;
(%o2)                              16
(%i3) x :bfloat(2/3);
(%o3)                           6.667b-1
(%i4) u : bfloat(2^(-18));
(%o4)                           3.815b-6
(%i5) x1 : x + u;
(%o5)                           6.667b-1
(%i6) x1 - x;
(%o6)                           1.526b-5
(%i7) x2 : x + 3.814b-6;
(%o7)                           6.667b-1
(%i8) x2 - x;
(%o8)                             0.0b0
(%i9) ulp : bfloat(2^(-17));
(%o9)                           7.629b-6
```

In computer science **Unit in the Last Place**, or **Unit of Least Precision**, **ulp(x)**, associated with a floating point number **x** is the gap between the two floating-point numbers closest to the value **x**. We assume here that the magnitude of **x** is less than 1. These two closest numbers will be $\mathbf{x} + \mathbf{u}$ and $\mathbf{x} - \mathbf{u}$ where $\mathbf{u}$ is the smallest positive floating point number which can be accurately represented by the systems of binary bits whose states are used to represent the fractional parts of the floating point numbers.

The amount of error in the evaluation of a floating-point operation is often expressed in ULP. We see that for **fpprec = 4**, 1 ULP is about $8 \cdot 10^{-6}$. An average error of 1 ULP is often seen as a tolerable error.

We can repeat this example for the case **fpprec = 16**.

```
(%i10) fpprec:16$
(%i11) ?fpprec;
(%o11)                              56
(%i12) x :bfloat(2/3);
(%o12)                     6.666666666666667b-1
(%i13) u : bfloat(2^(-58));
(%o13)                     3.469446951953614b-18
(%i14) x1 : x + u;
(%o14)                     6.666666666666667b-1
(%i15) x1 - x;
(%o15)                     1.387778780781446b-17
(%i16) x2 : x + 3.469446951953613b-18;
(%o16)                     6.666666666666667b-1
(%i17) x2 - x;
(%o17)                            0.0b0
(%i18) ulp : bfloat(2^(-57));
(%o18)                     6.938893903907228b-18
```

9.2.6 Effect of Floating Point Precision on Function Evaluation

Increasing the value of **fpprec** allows a more accurate numerical value to be found for the value of a function at some point. A simple function which allows one to find the absolute value of the change produced by increasing the value of **fpprec** has been presented by Richard Fateman.* This function is **uncert(f, arglist)**, in which **f** is a Maxima function, depending on one or more variables, and **arglist** is the n-dimensional point at which one wants the change in value of **f** produced by an increase of **fpprec** by **10**. This function returns a two element list consisting of the numerical value of the function at the requested point and also the absolute value of the difference induced by increasing the value of the current **fpprec** setting by the amount **10**.

We present here a version of Fateman's function which has an additional argument to control the amount of the increase of **fpprec**, and also has been simplified to accept only a function of one variable.

The function **fdf** is available in the Ch. 8 files **fdf.mac**, **qbromberg.mac**, **quad_ts.mac**, and **quad_de.mac**, and is defined by the code

```
fdf (%ff, %xx, %dfp) :=
   block([fv1,fv2,df],
    fv1:bfloat(%ff(bfloat(%xx))),
    block([fpprec:fpprec + %dfp ],
     fv2:  bfloat(%ff(bfloat(%xx))),
     df: abs(fv2 - fv1) ),
   [bfloat(fv2),bfloat(df)] )$
```

Here is an example of how this function can be used.

```
(%i1) (fpprintprec:8,load(fdf))$
(%i2) fpprec;
(%o2)                               16
(%i3) g(x) := sin(x/2)$
```

*see his draft paper "Numerical Quadrature in a Symbolic/Numerical Setting", Oct. 16, 2008, available as the file **quad.pdf** in the folder: **http://www.cs.berkeley.edu/~fateman/papers/**

```
(%i4) fdf(g,1,10);
(%o4)                    [4.7942553b-1, 1.834924b-18]
(%i5) fdf(g,1,10),fpprec:30;
(%o5)                    [4.7942553b-1, 2.6824592b-33]
(%i6) fpprec;
(%o6)                                16
```

In the first example, **fpprec** is **16**, and increasing the value to **26** produces a change in the function value of about 2×10^{-18}. In the second example, **fpprec** is **30**, and increasing the value to **40** produces a change in the function value of about 3×10^{-33}.

In the later section describing the "tanh-sinh" quadrature method, we will use this function for a heuristic estimate of the contribution of floating point errors to the approximate numerical value produced for an integral by that method.

9.3 Arbitrary Precision Quadrature with Maxima

9.3.1 Using bromberg for Arbitrary Precision Quadrature

A bigfloat version of the **romberg** function is defined in the file `brmbrg.lisp` located in `share/numeric`. You need to use **load(brmbrg)** or **load("brmbrg.lisp")** to use the function **bromberg**.

The use of **bromberg** is identical to the use of the **romberg** which we discussed in Chapter 8 (Numerical Integration), except that **rombergtol** (used for a relative error precision return) is replaced by the bigfloat **brombergtol** with a default value of **1.0b-4**, and **rombergabs** (used for an absolute error return) is replaced by the bigfloat **brombergabs** which has the default value **0.0b0**, and **rombergit** (which causes an return after halving the step size that many times) is replaced by the integer **brombergit** which has the default value **11**, and finally, **rombergmin** (the minimum number of halving iterations) is replaced by the integer **brombergmin** which has the default value **0**.

If the function being integrated has a magnitude of order one over the domain of integration, then an absolute error precision of a given size is approximately equivalent to a relative error precision of the same size. We will test **bromberg** using the function **exp(x)** over the domain **[-1, 1]**, and use only the absolute error precision parameter **brombergabs**, setting **brombergtol** to **0.0b0** so that the relative error test cannot be satisfied. Then the approximate value of the integral is returned when the absolute value of the change in value from one halving iteration to the next is less than the bigfloat number **brombergabs**.

We explore the use and behavior of **bromberg** for the simple integral $\int_{-1}^{1} e^x \, dx$, binding a value accurate to 42 digits to **tval**, defining parameter values, calling **bromberg** first with **fpprec** equal to 30 together with **brombergabs** set to **1.0b-15** and find an actual error (compared with **tval**) of about 7×10^{-24}.

```
(%i1) (fpprintprec:8,load(brmbrg));
(%o1) C:/PROGRA~1/MAXIMA~3.1/share/maxima/5.18.1/share/numeric/brmbrg.lisp
(%i2) [brombergtol,brombergabs,brombergit,brombergmin,fpprec,fpprintprec];
(%o2)                    [1.0b-4, 0.0b0, 11, 0, 16, 8]
(%i3) tval: bfloat(integrate(exp(x),x,-1,1)),fpprec:42;
(%o3)                             2.3504023b0
(%i4) fpprec;
(%o4)                                16
```

```
(%i5) (brombergtol:0.0b0,brombergit:100)$
(%i6) b15:(brombergabs:1.0b-15,bromberg(exp(x),x,-1,1) ),fpprec:30;
(%o6)                             2.3504023b0
(%i7) abs(b15 - tval),fpprec:42;
(%o7)                            6.9167325b-24
(%i8) b20:(brombergabs:1.0b-20,bromberg(exp(x),x,-1,1) ),fpprec:30;
(%o8)                             2.3504023b0
(%i9) abs(b20 - tval),fpprec:42;
(%o9)                            1.5154761b-29
```

We see that, for the case of this test integral involving a well behaved integrand, the actual error of the result returned by **bromberg** is much smaller than the requested "difference error" supplied by the parameter **brombergabs**.

For later use, we define **qbromberg** (in a file **qbromberg.mac**) with the code:

```
qbromberg(%f,a,b,rprec,fp, itmax ) :=
        block([brombergtol,brombergabs,brombergit,
                   fpprec:fp ],
          if rprec > fp then
           ( print(" rprec should be less than fp "),
             return(done) ),
          brombergabs : bfloat(10^(-rprec)),
          brombergtol : 0.0b0,
          brombergit : itmax,
          bromberg(%f(x),x,a,b) )$
```

This function, with the syntax

```
    qbromberg ( f, a, b, rprec, fp, itmax )
```

uses the Maxima function `bromberg` to integrate the Maxima function `f` over the interval `[a, b]`, setting the local value of `fpprec` to `fp`, setting `brombergtol` to `0`, setting `brombergabs` to $10^{-\mathrm{rprec}}$, where `rprec` is called the "requested precision".

Here is a test of **qbromberg** for this simple integral.

```
(%i10) load(qbromberg)$
(%i11) qbr20 : qbromberg(exp,-1,1,20,40,100);
(%o11)                            2.3504023b0
(%i12) abs(qbr20 - tval);
(%o12)                              0.0b0
(%i13) abs(qbr20 - tval),fpprec:40;
(%o13)                          1.0693013b-29
```

We have to be careful in the above step-by-step method to set **fpprec** to a large enough value to see the actual size of the error in the returned answer.

Instead of the work involved in the above step by step method, it is more convenient to define a function **qbrlist** which is passed a desired **fpprec** as well as a list of requested precision goals for **bromberg**. The function **qbrlist** then assumes a sufficiently accurate **tval** is globally defined, and proceeds through the list to calculate the **bromberg** value for each requested precision, computes the error in the result, and prints a line containing (rprec, fpprec, value, value-error). Here is the code for such a function, available in **qbromberg.mac**:

```
qbrlist(%f,a,b,rplist,fp,itmax) :=
  block([fpprec:fp,fpprintprec,brombergtol,
              brombergabs,brombergit,val,verr,pr],
     if not listp(rplist) then (print("rplist # list"),return(done)),
     brombergtol : 0.0b0,
     brombergit : itmax,
     fpprintprec:8,
     print(" rprec   fpprec     val              verr "),
     print(" "),
     for pr in rplist do
      ( brombergabs : bfloat(10^(-pr)),
        val: bromberg(%f(x),x,a,b),
        verr: abs(val - tval),
        print("   ",pr,"    ",fp,"    ",val,"    ",verr) ) )$
```

and here is an example of use of **qbrlist** in which the requested precision **rprec** (called **pr** in the code) is set to three different values supplied by the list **rplist** for each setting of **fpprec** used. We first define an accurate comparison value **tval**:

```
(%i1) (fpprintprec:8, load(brmbrg), load(qbromberg))$
(%i2) tval: bfloat(integrate(exp(x),x,-1,1)),fpprec:42;
(%o2)                             2.3504023b0
```

Here is our test for three different values of **fpprec**:

```
(%i3) qbrlist(exp,-1,1,[10,15,17 ],20,100)$
 rprec   fpprec      val              verr
   10      20      2.3504023b0      4.5259436b-14
   15      20      2.3504023b0      1.3552527b-20
   17      20      2.3504023b0      1.3552527b-20
(%i4) qbrlist(exp,-1,1,[10,20,27 ],30,100)$
 rprec   fpprec      val              verr
   10      30      2.3504023b0      4.5259437b-14
   20      30      2.3504023b0      1.4988357b-29
   27      30      2.3504023b0      5.5220263b-30
(%i5) qbrlist(exp,-1,1,[10,20,30,35],40,100)$
 rprec   fpprec      val              verr
   10      40      2.3504023b0      4.5259437b-14
   20      40      2.3504023b0      1.0693013b-29
   30      40      2.3504023b0      1.1938614b-39
   35      40      2.3504023b0      1.1938614b-39
```

We see that with **fpprec** equal to **40**, increasing **rprec** from **30** to **35** results in no improvement in the actual error of the result.

When bromberg Fails

If the integrand has end point algebraic and/or logarithmic singularities, **bromberg** may fail. Here is an example in which the integrand has a logarithmic singularity at the lower end point: $\int_0^1 \sqrt{t}\ln(t)\,dt$. The **integrate** function has no problem with this integral.

```
(%i6) g(x):= sqrt(x)*log(x)$
(%i7) integrate(g(t),t,0,1);
                                                4
(%o7)                                         - -
                                                9
(%i8) (load(brmbrg),load(qbromberg))$
(%i9) qbromberg(g,0,1,30,40,100);
log(0) has been generated.
#0: qbromberg(%f=g,a=0,b=1,rprec=30,fp=40,itmax=100)
 -- an error.  To debug this try debugmode(true);
```

You can instead use the tanh-sinh quadrature method for this integral (see Sec. 9.3.3).

9.3.2 A Double Exponential Quadrature Method for $a \le x < \infty$

This method (H. Takahasi and M. Mori, 1974; see Sec 9.3.3) is effective for integrands which contain a factor with some sort of exponential damping as the integration variable becomes large.

An integral of the form $\int_a^\infty g(y)\,dy$ can be converted into the integral $\int_0^\infty f(x)\,dx$ by making the change of variable of integration $y \to x$ given by $y = x + a$. Then $f(x) = g(x + a)$.

The double exponential method used here then converts the integral $\int_0^\infty f(x)\,dx$ into the integral $\int_{-\infty}^\infty F(u)\,du$ using a variable transformation $x \to u$:

$$x(u) = \exp(u - \exp(-u)) \tag{9.2}$$

and hence

$$F(u) = f(x(u))\,w(u), \qquad \text{where} \qquad w(u) = \frac{dx}{du} = \exp(-\exp(-u)) + x(u). \tag{9.3}$$

You can confirm that $x(0) = \exp(-1)$, $\quad w(0) = 2\,x(0)$ and that $x(-\infty) = 0$, $\quad x(\infty) = \infty$.

Because of the rapid decay of the integrand when the magnitude of u is large, one can approximate the value of the infinite domain u integral by using a trapezoidal numerical approximation with step size h using a modest number $(2\,N + 1)$ of function evaluations.

$$I(h, N) \simeq h \sum_{j=-N}^{N} F(u_j) \quad \text{where} \quad u_j = j\,h \tag{9.4}$$

This method is implemented in the Ch. 8 file **quad_de.mac**. We first demonstrate the available functions on the simple integral $\int_0^\infty e^{-x}\,dx = 1$.

```
(%i1) fpprintprec:8$
(%i2) g(x):= exp(-x)$
(%i3) tval : bfloat(integrate(g(x),x,0,inf)),fpprec:45;
(%o3)                               1.0b0
```

```
(%i4) load(quad_de);
(%o4)                          c:/work3/quad_de.mac
(%i5) quad_de(g,0,30,40);
(%o5)                        [1.0b0, 4, 4.8194669b-33]
(%i6) abs(first(%) - tval),fpprec:45;
(%o6)                                9.1835496b-41
```

The package function **quad_de(f, a, rp, fp)** integrates the Maxima function **f** over the domain $[x \geq a]$, using **fpprec : fp**, and returns a three element list when **vdiff** (the absolute value of the difference obtained for the integral in successive k levels) becomes less than or equal to 10^{-rp}. The parameter **rp** is called the "requested precision", and the value of **h** is repeatedly halved until the **vdiff** magnitude either satisfies this criterion or starts increasing. The first element is the appoximate value of the integral. The second element (4 above) is the "final k-level" used, where $h = 2^{-k}$. The third and last element is the final value of **vdiff**. We see in the above example that requesting precision $rp = 30$ and using floating point precision **fpprec : 40** results in an answer good to about **40** digits. This sort of accuracy is typical.

The package function **idek(f, a, k, fp)** integrates the Maxima function **f** over the domain $[a, \infty]$ using a "k-level approximation" with $h = 1/2^k$ and **fpprec : fp**.

```
(%i7) idek(g,0,4,40);
(%o7)                                    1.0b0
(%i8) abs(% - tval),fpprec:45;
(%o8)                                9.1835496b-41
```

The package function **idek_e(f, a, k, fp)** does the same calculation as **idek(f, a, k, fp)**, but returns both the approximate value of the integral and also a rough estimate of the amount of the error which is due to the floating point arithmetic precision being used. (The error of the approximation has three contributions: 1. the quadrature algorithm being used, 2. the step size **h** being used, and 3. the precision of the floating point arithmetic being used.)

```
(%i9) idek_e(g,0,4,40);
(%o9)                        [1.0b0, 8.3668155b-42]
(%i10) abs(first(%) - tval),fpprec:45;
(%o10)                               9.1835496b-41
```

The package function **ide(f, a, rp, fp)** follows the same path as **quad_de(f, a, rp, fp)**, but shows the progression toward success as the k level increases (and **h** decreases):

```
(%i11) ide(g,0,30,40);
      rprec =  30  fpprec =  40
 k     value          vdiff
 1    1.0b0
 2    1.0b0          4.9349774b-8
 3    9.9999999b-1   4.8428706b-16
 4    1.0b0          4.8194669b-33
```

The package function **`ide_test(f, a, rp, fp)`** follows the path of **`ide(f, a, rp, fp)`**, but adds to the table the value of the error of the approximate result for each **k** level attempted. The use of this function depends on an accurate value of the integral being bound to the global variable **tval**.

```
(%i12) ide_test(g,0,30,40);
     rprec =  30  fpprec =  40
 k     value           vdiff             verr
 1    1.0b0                            4.9349775b-8
 2    1.0b0          4.9349774b-8     4.8428706b-16
 3    9.9999999b-1   4.8428706b-16    4.8194668b-33
 4    1.0b0          4.8194669b-33    9.1835496b-41
```

Test Integral 1

Here we test this double exponential method code with the known integral

$$\int_0^\infty \frac{e^{-t}}{\sqrt{t}}\, dt = \sqrt{\pi} \tag{9.5}$$

```
(%i13) g(x):= exp(-x)/sqrt(x)$
(%i14) integrate(g(t),t,0,inf);
(%o14)                              sqrt(%pi)
(%i15) tval : bfloat(%),fpprec:45;
(%o15)                             1.7724538b0
(%i16) quad_de(g,0,30,40);
(%o16)                  [1.7724538b0, 4, 1.0443243b-34]
(%i17) abs(first(%) - tval),fpprec:45;
(%o17)                            1.8860005b-40
(%i18) idek_e(g,0,4,40);
(%o18)                   [1.7724538b0, 2.7054206b-41]
```

Again we see that the combination **rp = 30, fp = 40** leads to an answer good to about **40** digits of precision.

Test Integral 2

Our second known integral is

$$\int_0^\infty e^{-t^2/2}\, dt = \sqrt{\pi/2} \tag{9.6}$$

```
(%i19) g(x) := exp(-x^2/2)$
(%i20) tval : bfloat(sqrt(%pi/2)),fpprec:45$
(%i21) quad_de(g,0,30,40);
(%o21)                  [1.2533141b0, 5, 1.099771b-31]
(%i22) abs(first(%) - tval),fpprec:45;
(%o22)                            2.1838045b-40
(%i23) idek_e(g,0,5,40);
(%o23)                   [1.2533141b0, 1.3009564b-41]
```

Test Integral 3

Our third test integral is

$$\int_{0}^{\infty} e^{-t} \cos t \, dt = 1/2 \tag{9.7}$$

```
(%i24) g(x) := exp(-x)*cos(x)$
(%i25) integrate(g(x),x,0,inf);
                                        1
(%o25)                                  -
                                        2
(%i26) tval : bfloat(%),fpprec:45$
(%i27) quad_de(g,0,30,40);
(%o27)                    [5.0b-1, 5, 1.7998243b-33]
(%i28) abs(first(%) - tval),fpprec:45;
(%o28)                          9.1835496b-41
(%i29) idek_e(g,0,5,40);
(%o29)                      [5.0b-1, 9.8517724b-42]
```

9.3.3 The tanh-sinh Quadrature Method for $a \le x \le b$

H. Takahasi and M. Mori (1974: see references at the end of this section) presented an efficient method for the numerical integration of the integral of a function over a finite domain. This method is known under the names "tanh-sinh method" and "double exponential method". This method can handle integrands which have algebraic and logarithmic end point singularities, and is well suited for use with arbitrary precision work.

Quoting (loosely) David Bailey's (see references below) slide presentations on this subject:

> The tanh-sinh quadrature method can accurately handle all "reasonable functions", even those with "blow-up singularities" or vertical slopes at the end points of the integration interval. In many cases, reducing the step size h by half doubles the number of correct digits in the result returned ("quadratic convergence").

An integral of the form $\int_a^b g(y)\, dy$ can be converted into the integral $\int_{-1}^{1} f(x)\, dx$ by making the change of variable of integration $y \to x$ given by $y = \alpha\, x + \beta$ with $\alpha = (b - a)/2$ and $\beta = (a + b)/2$. Then $f(x) = \alpha\, g(\alpha\, x + \beta)$.

The tanh-sinh method introduces a change of variables $x \to u$ which implies

$$\int_{-1}^{1} f(x)\, dx = \int_{-\infty}^{\infty} F(u)\, du. \tag{9.8}$$

The change of variables is expressed by

$$x(u) = \tanh\left(\frac{\pi}{2} \sinh u\right) \tag{9.9}$$

and you can confirm that

$$u = 0 \Rightarrow x = 0, \qquad u \to -\infty \Rightarrow x \to -1, u \to \infty \Rightarrow x \to 1 \tag{9.10}$$

We also have $x(-u) = -x(u)$.

The "weight" $\mathbf{w(u) = dx(u)/du}$ is

$$\mathbf{w(u)} = \frac{\frac{\pi}{2}\,\mathbf{\cosh u}}{\mathbf{\cosh^2\left(\frac{\pi}{2}\sinh u\right)}} \tag{9.11}$$

with the property $\mathbf{w(-u) = w(u)}$, in terms of which $\mathbf{F(u) = f(x(u)\,w(u)}$. Moreover, $\mathbf{F(u)}$ has "double exponential behavior" of the form

$$\mathbf{F(u)} \approx \exp\left(-\frac{\pi}{2}\exp(\,|\,\mathbf{u}\,|\,)\right) \qquad \text{for} \quad \mathbf{u} \to \pm\infty. \tag{9.12}$$

Because of the rapid decay of the integrand when the magnitude of $\mathbf{u}$ is large, one can approximate the value of the infinite domain integral by using a trapezoidal numerical approximation with step size $\mathbf{h}$ using a modest number $(\mathbf{2\,N+1})$ of function evaluations.

$$\mathbf{I(h,N)} \simeq \mathbf{h}\sum_{\mathbf{j=-N}}^{\mathbf{N}} \mathbf{F(u_j)} \quad \text{where} \quad \mathbf{u_j = j\,h} \tag{9.13}$$

This method is implemented in the Ch.8 file **quad_ts.mac** and we will illustrate the available functions using the simple integral $\int_{-1}^{1} \mathrm{e}^{\mathrm{x}}\,\mathrm{dx}$.

The package function **quad_ts (f, a, b, rp, fp)** is the most useful workhorse for routine use, and uses the tanh-sinh method to integrate the Maxima function **f** over the finite interval **[a, b]**, stopping when the absolute value of the difference $(\mathbf{I_k - I_{k-1}})$ is less than $\mathbf{10^{-rp}}$ (rp is the "requested precision" for the result), using **fp** digit precision arithmetic (**fpprec** set to **fp**, and **bfloat** being used to enforce this arithmetic precision). This function returns the list
[approx-value, k-level-used, abs(vdiff)], where the last element should be smaller than $\mathbf{10^{-rp}}$.

```
(%i1) fpprintprec:8$
(%i2) tval : bfloat( integrate( exp(x),x,-1,1 )),fpprec:45;
(%o2)                          2.3504023b0
(%i3) load(quad_ts);
   _kmax% =  8    _epsfac% =  2
(%o3)                     c:/work3/quad_ts.mac
(%i4) bfprint(tval,45)$
   number of digits =  45
   2.35040238728760291376476370119120163031143596b0
(%i5) quad_ts(exp,-1,1,30,40);
 construct _yw%[kk,fpprec] array for kk =  8  and fpprec =  40   ...working...
(%o5)                    [2.3504023b0, 5, 0.0b0]
(%i6) abs(first(%) - tval),fpprec:45;
(%o6)                          2.719612b-40
```

A value of the integral accurate to about 45 digits is bound to the symbol **tval**. The package function **bfprint(bf, fpp)** allows controlled printing of **fpp** digits of the "true value" **tval** to the screen. We then compare the approximate quadrature result with this "true value". The package **quad_ts.mac** defines two global parameters. **_kmax%** is the maximum "k-level" possible (the defined default is 8, which means the minimum step size for the transformed "u-integral" is $\mathbf{du = h = 1/2^8 = 1/256}$. The actual "k-level" needed to return a result with the requested precision **rp** is the integer in the second element of the returned list. The global parameter **_epsfac%** (default value **2**) is used to decide how many **(y, w)** numbers to pre-compute (see below).

We see that a "k-level" approximation with $k = 5$ and $h = 1/2^5 = 1/32$ returned an answer with an actual precision of about 40 digits (when $rp = 30$ and $fp = 40$).

The first time 40 digit precision arithmetic is called for, a set of (y, w) numbers are calculated and stored in an array which we call **`_yw%[8, 40]`**. The **`y(u)`** values will later be converted to **`x(u)`** numbers using high precision, and the original integrand function **`f(x(u))`** is also calculated at high precision. The **`w(u)`** numbers are what we call "weights", and are needed for the numbers $F(u) = f(x(u))\,w(u)$ used in the trapezoidal rule evaluation. The package precomputes pairs **`(y, w)`** for larger and larger values of **`u`** until the magnitude of the weight **`w`** becomes less than **`eps`**, where $eps = 10^{-np}$, where **`n`** is the global parameter **`_epsfac%`** (default **`2`**) and **`p`** is the requested floating point precision **`fp`**.

Once the set of **`40-digit`** precision **`(y, w)`** numbers have been "pre-computed", they can be used for the evaluation of any similar precision integrals later, since these numbers are independent of the actual function being integrated, but depend only on the nature of the tanh-sinh transformation being used.

The package function **`qtsk(f, a, b, k, fp)`** (note: arg **k** replaces **rp**) integrates the Maxima function **`f`** over the domain $[a, b]$ using a "k-level approximation" with $h = 1/2^k$ and **`fpprec : fp`**.

```
(%i7) qtsk(exp,-1,1,5,40);
(%o7)                            2.3504023b0
(%i8) abs(% - tval),fpprec:45;
(%o8)                           2.719612b-40
```

A heuristic value of the error contribution due to the arithmetic precision being used (which is separate from the error contribution due to the nature of the algorithm and the step size being used) can be found by using the package function **`qtsk_e(f, a, b, k, fp);`**. The first element of the returned list is the value of the integral, the second element of the returned list is a rough estimate of the contribution of the floating point arithmetic precision being used to the error of the returned answer.

```
(%i9) qtsk_e(exp,-1,1,5,40);
(%o9)                    [2.3504023b0, 2.0614559b-94]
(%i10) abs(first(%) - tval),fpprec:45;
(%o10)                          2.719612b-40
```

The very small estimate of the arithmetic precision contribution (two parts in 10^{94}) to the error of the answer is due to the high precision being used to convert from the pre-computed **y** to the needed abcissa **x** via $x : \mathrm{bfloat}(1 - y)$ and the subsequent evaluation $f(x)$. The precision being used depends on the size of the smallest **y** number, which will always be that appearing in the last element of the hashed array **`_yw%[8, 40]`**.

```
(%i11) last(_yw%[8,40]);
(%o11)                  [4.7024891b-83, 8.9481574b-81]
```

(In Eq. (9.13) we have separated out the **(u = 0, x = 0)** term, and used the symmetry properties $x(-u) = -x(u)$, and $w(-u) = w(u)$ to write the remainder as a sum over positive values of **u** (and hence positive values of **x**) so only the large **u** values of **y(u)** need to be pre-computed).

We see that the smallest **y** number is about 5×10^{-83} and if we subtract this from **`1`** we will get **`1`** unless we use a very high precision. It turns out that as **u** approaches plus infinity, **x** (as used here) approaches **b** (which is **`1`** in our example) from values less than **b**. Since a principal virtue of the tanh-sinh method is its ability to handle integrands which "blow up" at the limits of integration, we need to make sure we stay away (even if

only a little) from those end limits.

We can see the precision with which the arithmetic is being carried out in this crucial step by using the **fpxy(fp)** function

```
(%i12) fpxy(40)$
 the last y value =  4.7024891b-83
 the fpprec being used for x and f(x) is  93
```

and this explains the small number returned (as the second element) by `qtsk_e(exp, -1, 1, 5, 40);`.

The package function `qts(f, a, b, rp, fp)` follows the same path as `quad_ts(f, a, b, rp, fp)`, but shows the progression toward success as the **k** level increases (and h decreases):

```
(%i13) qts(exp,-1,1,30,40)$
  rprec =  30   fpprec =  40
  k        newval          vdiff
  1     2.350282b0
  2     2.3504023b0    1.2031242b-4
  3     2.3504023b0    8.136103b-11
  4     2.3504023b0    1.9907055b-23
  5     2.3504023b0    0.0b0
```

The package function `qts_test(f, a, b, rp, fp)` follows the path of `qts(f, a, b, rp, fp)`, but adds to the table the value of the error of the approximate result for each **k** level attempted. The use of this function depends on an accurate value of the integral being bound to the global variable **tval**.

```
(%i14) qts_test(exp,-1,1,30,40)$
      rprec =  30  fpprec =  40
 k      value          vdiff              verr
 1     2.350282b0                       1.2031234b-4
 2     2.3504023b0    1.2031242b-4      8.136103b-11
 3     2.3504023b0    8.136103b-11      1.9907055b-23
 4     2.3504023b0    1.9907055b-23     2.7550648b-40
 5     2.3504023b0    0.0b0             2.7550648b-40
```

Test Integral 1

Here we test this tanh-sinh method code with the known integral which confounded **bromberg** in Sec. 9.3.1 :

$$\int_0^1 \sqrt{t}\ln(t)\,dt = -4/9 \tag{9.14}$$

```
(%i15) g(x):= sqrt(x)*log(x)$
(%i16) tval : bfloat(integrate(g(t),t,0,1)),fpprec:45;
(%o16)                          - 4.4444444b-1
(%i17) quad_ts(g,0,1,30,40);
(%o17)                 [- 4.4444444b-1, 5, 3.4438311b-41]
(%i18) abs(first(%) - tval),fpprec:45;
(%o18)                           4.4642216b-41
(%i19) qtsk_e(g,0,1,5,40);
(%o19)                   [- 4.4444444b-1, 7.6556481b-43]
```

Requesting thirty digit accuracy with forty digit arithmetic returns a value for this integral which has about forty digit precision. Note that “vdiff” is approximately the same as the actual absolute error.

Test Integral 2

Consider the integral

$$\int_0^1 \frac{\arctan(\sqrt{2+t^2})}{(1+t^2)\,\sqrt{2+t^2}}\,dt = 5\,\pi^2/96. \tag{9.15}$$

```
(%i20) g(x):= atan(sqrt(2+x^2))/(sqrt(2+x^2)*(1+x^2))$
(%i21) integrate(g(t),t,0,1);
                                 1
                                /              2
                                [    atan(sqrt(t  + 2))
(%o21)                          I   --------------------- dt
                                ]     2             2
                                /   (t  + 1) sqrt(t  + 2)
                                 0
(%i22) quad_qags(g(t),t,0,1);
(%o22)                    [0.514042, 5.70701148E-15, 21, 0]
(%i23) float(5*%pi^2/96);
(%o23)                                0.514042
(%i24) tval: bfloat(5*%pi^2/96),fpprec:45;
(%o24)                              5.1404189b-1
(%i25) quad_ts(g,0,1,30,40);
(%o25)                    [5.1404189b-1, 5, 1.5634993b-36]
(%i26) abs(first(%) - tval),fpprec:45;
(%o26)                              7.3300521b-41
(%i27) qtsk_e(g,0,1,5,40);
(%o27)                      [5.1404189b-1, 1.3887835b-43]
```

Test Integral 3

We consider the integral

$$\int_0^1 \frac{\sqrt{t}}{\sqrt{1-t^2}}\,dt = 2\,\sqrt{\pi}\,\Gamma(3/4)/\Gamma(1/4) \tag{9.16}$$

```
(%i28) g(x):= sqrt(x)/sqrt(1 - x^2)$
(%i29) quad_qags(g(t),t,0,1);
(%o29)                   [1.1981402, 8.67914629E-11, 567, 0]
(%i30) integrate(g(t),t,0,1);
                                             1   3
                                        beta(-, -)
                                             2   4
(%o30)                                  ----------
                                            2
(%i31) tval : bfloat(%),fpprec:45;
(%o31)                                1.1981402b0
(%i32) quad_ts(g,0,1,30,40);
(%o32)                     [1.1981402b0, 5, 1.3775324b-40]
(%i33) abs(first(%) - tval),fpprec:45;
(%o33)                              1.8628161b-40
(%i34) qtsk_e(g,0,1,5,40);
(%o34)                      [1.1981402b0, 1.5833892b-45]
```

An alternative route to the “true value” is to convert **beta** to **gamma**’s using **makegamma**:

```
(%i35) makegamma(%o30);
                                                 3
                             2 sqrt(%pi)  gamma(-)
                                                 4
(%o35)                       --------------------
                                          1
                                    gamma(-)
                                          4
(%i36) float(%);
(%o36)                             1.1981402
(%i37) bfloat(%o11),fpprec:45;
(%o37)                            1.1981402b0
```

Test Integral 4

We next consider the integral

$$\int_0^1 \ln^2 t \, dt = 2 \tag{9.17}$$

```
(%i38) g(x) := log(x)^2$
(%i39) integrate(g(t),t,0,1);
(%o39)                                 2
(%i40) quad_ts(g,0,1,30,40);
(%o40)                         [2.0b0, 5, 0.0b0]
(%i41) abs( first(%) - bfloat(2) ),fpprec:45;
(%o41)                           1.8367099b-40
(%i42) qtsk_e(g,0,1,5,40);
(%o42)                      [2.0b0, 1.2570464b-42]
```

Test Integral 5

We finally consider the integral

$$\int_0^{\pi/2} \ln(\cos t) \, dt = -\pi \ \ln(2)/2 \tag{9.18}$$

```
(%i43) g(x) := log( cos(x) )$
(%i44) quad_qags(g(t),t,0,%pi/2);
(%o44)               [- 1.088793, 1.08801856E-14, 231, 0]
(%i45) integrate(g(t),t,0,%pi/2);
                                %pi
                                ---
                                 2
                               /
                               [
(%o45)                         I    log(cos(t)) dt
                               ]
                               /
                                0
```

```
(%i46) float(-%pi*log(2)/2);
(%o46)                           - 1.088793
(%i47) tval : bfloat(-%pi*log(2)/2),fpprec:45;
(%o47)                          - 1.088793b0
(%i48) quad_ts(g,0,%pi/2,30,40);
(%o48)         [1.2979374b-80 %i - 1.088793b0, 5, 9.1835496b-41]
(%i49) ans: realpart( first(%) );
(%o49)                          - 1.088793b0
(%i50) abs(ans - tval),fpprec:45;
(%o50)                          1.9661653b-40
(%i51) qtsk_e(g,0,%pi/2,5,40);
(%o51)          [1.2979374b-80 %i - 1.088793b0, 2.4128523b-42]
```

We see that the tanh-sinh result includes a tiny imaginary part due to bigfloat errors, and taking the real part produces an answer good to about 40 digits (using `rp = 30, fp = 40`).

References for the tanh-sinh Quadrature Method

This method was initially described in the article **Double Exponential Formulas for Numerical Integration**, by Hidetosi Takahasi and Masatake Mori, in the journal Publications of the Research Institute for Mathematical Sciences (Publ. RIMS), vol.9, Number 3, (1974), 721-741, Kyoto University, Japan. A recent summary by the second author is **Discovery of the Double Exponential Transformation and Its Developments**, by Masatake Mori, Publ. RIMS, vol.41, Number 4, (2005), 897-935. Both of the above articles can be downloaded from the Project Euclid RIMS webpage

```
http://projecteuclid.org/
              DPubS?service=UI&version=1.0&verb=Display&page=past&handle=euclid.prims
```

A good summary of implementation ideas can be found in the report **Tanh-Sinh High-Precision Quadrature**, by David H. Bailey, Jan. 2006, LBNL-60519, which can be downloaded from the webpage

```
  http://crd.lbl.gov/~dhbailey/dhbpapers/
```

Further Improvements for the tanh-sinh Quadrature Method

The code provided in the file **`quad_ts.mac`** has been lightly tested, and should be used with caution.

No proper investigation has been made of the efficiency of choosing to use a floating point precision (for all terms of the sum) based on the small size of the smallest `y` value.

No attempt has been made to translate into Lisp and compile the code to make timing trials for comparison purposes.

These (and other) refinements are left to the initiative of the hypothetical alert reader of limitless dedication (HAROLD).

9.3.4 The Gauss-Legendre Quadrature Method for $a \le x \le b$

Loosely quoting from David Bailey's slide presentations (see references at the end of the previous section)

> The Gauss-Legendre quadrature method is an efficient method for continuous, well-behaved functions. In many cases, doubling the number of points at which the integrand is evaluated doubles the number of correct digits in the result. This method performs poorly for functions with algebraic and/or logarithmic end point singularities. The cost of computing the zeros of the Legendre polynomials and the corresponding "weights" increases as n^2 and thus becomes impractical for use beyond a few hundred digits.

Since one can always make a change of integration variable from the domain **[a, b]** to the integration domain **[-1, 1]**, this method approximates an integral over **[-1, 1]** as the sum

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{j=1}^{N} w_j\, f(x_j) \tag{9.19}$$

where the x_j are the roots of the N-th degree Legendre polynomial $P_N(x)$ on **[-1, 1]**, and the weights w_j are

$$w_j = \frac{-2}{(N+1)\,P'_N(x_j)\,P_{N+1}(x_j)} \tag{9.20}$$

This method is implemented with our Ch.9 package file **quad_gs.mac**, and Richard Fateman's lisp file:

http://www.cs.berkeley.edu/˜fateman/generic/quad-maxima.lisp

which should be downloaded to use this package.

Except for the "driver" Maxima function **quad_gs(f,a,b,rp)**, all the Maxima functions in **quad_gs.mac** are functions defined by Fateman in either the comment section of the above lisp file, or in the file **http://www.cs.berkeley.edu/˜fateman/papers/quadmax.mac**.

An introduction to arbitrary precision code and some background to the problem of arbitrary precision quadrature has been provided by Richard Fateman in his draft paper **Numerical Quadrature in a Symbolic/Numerical Setting**, **quad.pdf** (see Sec. 9.2.6).

We will illustrate the available functions using the simple integral $\int_{-1}^{1} e^x\,dx$.

The package function **gaussunit(f, N)** integrates the Maxima function **f** over the domain $[-1, 1]$ using N point Gauss-Legendre quadrature.

```
(%i1) load("quad-maxima.lisp");
(%o1)                          quad-maxima.lisp
(%i2) fpprintprec:8$
(%i3) tval : bfloat(integrate(exp(x),x,-1,1)),fpprec:45;
(%o3)                             2.3504023b0
(%i4) load(quad_gs);
(%o4)                        c:/work3/quad_gs.mac
(%i5) arrays;
(%o5)                            [ab_and_wts]
(%i6) arrayinfo(ab_and_wts);
(%o6)                             [hashed, 2]
(%i7) gaussunit(exp,4);
(%o7)                             2.350402b0
(%i8) abs(% - tval),fpprec:45;
(%o8)                            2.9513122b-7
(%i9) fpprec;
(%o9)                                 16
(%i10) arrayinfo(ab_and_wts);
(%o10)                       [hashed, 2, [4, 16]]
(%i11) first( ab_and_wts[4, 16] );
(%o11)                  [8.6113631b-1, 3.3998104b-1]
(%i12) second( ab_and_wts[4, 16] );
(%o12)                  [3.4785484b-1, 6.5214515b-1]
```

```
(%i13) lp4 : legenp(4,x);
                                   4         2
                               35 x      15 x     3
(%o13)                         ----- - ----- + -
                                 8         4      8
(%i14) float(solve(lp4));
(%o14)   [x = - 0.861136, x = 0.861136, x = - 0.339981, x = 0.339981]
```

With the default value of **fpprec = 16** and using only four integrand evaluation points, the error is about 2 parts in 10^7. The first element of the two index hashed array **ab_and_wts[4,16]** is a list of the positive zeros of the fourth order Legendre polynomial $P_4(x)$ (calculated with the arithmetic precision **fpprec = 16**).

That fourth order Legendre polynomial $P_4(x)$ can be displayed with this package using **legenp(4, x)**. Using **solve** we see that the roots for negative x are simply the the positive roots with a minus sign, so the algorithm used makes use of this symmetry and keeps track of only the positive roots.

We can verify that the list of roots returned is correct to within the global floating point precision (we do this two different ways):

```
(%i15) lfp4(x) := legenp(4,x)$
(%i16) map('lfp4,%o11);
(%o16)                      [0.0b0, - 3.4694469b-18]
(%i17) map( lambda([z],legenp(4,z)),%o11 );
(%o17)                      [0.0b0, - 3.4694469b-18]
```

The second element of **ab_and_wts[4,16]** is a list of the weights which are associated with the positive roots (the negative roots have the same weights), with order corresponding to the order of the returned positive roots.

The package function **gaussunit_e(f, N)** does the same job as **gaussunit(f, N)**, but returns a rough estimate of the amount contributed to the error by the floating point precision used (as the second element of a list:

```
(%i18) gaussunit_e(exp,4);
(%o18)                     [2.350402b0, 1.2761299b-17]
```

We see that the error attributable to the floating point precision used is insignificant compared to the error due to the low number of integrand evaluation points for this example.

An arbitrary finite integration interval is allowed with the functions **gaussab(f, a, b, N)** and **gaussab_e(f, a, b, N)** which use **N** point Gauss-Legendre quadrature over the interval $[a, b]$, with the latter function being the analog of **gaussunit_e(f, N)**.

```
(%i19) gaussab(exp,-1,1,4);
(%o19)                              2.350402b0
(%i20) abs(% - tval),fpprec:45;
(%o20)                             2.9513122b-7
(%i21) gaussab_e(exp,-1,1,4);
(%o21)                     [2.350402b0, 1.2761299b-17]
```

The package function **quad_gs (f, a, b, rp)** integrates the Maxima function **f** over the finite interval $[a, b]$, successively doubling the number of integrand evaluation points, stopping when the absolute value of the difference $(I_n - I_{n/2})$ is less than 10^{-rp} (**rp** is the "requested precision" for the result), using the global setting of **fpprec** to use the corresponding precision arithmetic. We emphasize that Fateman's code uses a global setting of **fpprec** to achieve higher precision quadrature, rather than the method used in the previous two sections in which **fpprec** was set "locally" inside a **block**. This function returns the list **[approx-value, number-function-evaluations, abs(vdiff)]**, where the last element should be smaller than 10^{-rp}.

Here we test this function for **fpprec = 16, 30, and 40**.

```
(%i22) quad_gs(exp,-1,1,10);
                                   fpprec = 16

(%o22)                 [2.3504023b0, 20, 6.6613381b-16]
(%i23) abs(first(%) -tval),fpprec:45;
(%o23)                            6.2016267b-16
(%i24) fpprec:30$
(%i25) quad_gs(exp,-1,1,20);
                                   fpprec = 30

(%o25)                 [2.3504023b0, 20, 1.2162089b-24]
(%i26) abs(first(%) -tval),fpprec:45;
(%o26)                            2.2001783b-30
(%i27) fpprec:40$
(%i28) quad_gs(exp,-1,1,30);
                                   fpprec = 40

(%o28)                 [2.3504023b0, 40, 1.8367099b-40]
(%i29) abs(first(%) -tval),fpprec:45;
(%o29)                            2.7905177b-40
(%i30) gaussab_e(exp,-1,1,40);
(%o30)                   [2.3504023b0, 1.8492214b-41]
```

We have checked the contribution to the error due to the forty digit arithmetic precision used, with $N = 40$ point Gauss-Legendre quadrature (remember that **N** is the middle element of the list returned by **quad_gs (f, a, b, rp)** and is also the last argument of the function **gaussab_e(f, a, b, N)**.

We see that requesting **30** digit precision for the answer while using the global **fpprec** set to **40** results in an answer good to about **39** digits. Finally, let's check on what abscissae and weight arrays have been calculated so far:

```
(%i31) arrayinfo(ab_and_wts);
(%o31) [hashed, 2, [4, 16], [10, 16], [10, 30], [10, 40], [20, 16], [20, 30],
                                                              [20, 40], [40, 40]]
```

Using **quad_gs (f, a, b, rp)** with **fpprec = 16** led to the calculation of the abscissae and weight array for the index pairs **[10,16]** and **[20,16]** before the requested precision was achieved (the function always starts with **N = 10** point quadrature and then successively doubles that number until success is achieved).

Using **quad_gs (f, a, b, rp)** with **fpprec = 30** led to the calculation of the abscissae and weight array for the index pairs **[10,30]** and **[20,30]** before the requested precision was achieved.

Using **quad_gs (f, a, b, rp)** with **fpprec = 40** led to the calculation of the abscissae and weight array for the index pairs **[10,40]**, **[20,40]**, and **[40,40]** before the requested precision was achieved.

Finally, we have the function **quad_gs_table(f, a, b, rp)** which prints out a table showing the progression toward success:

```
(%i32) quad_gs_table(exp,-1,1,30)$
                                  fpprec = 40
      new val         N           vdiff
   2.3504023b0       10
   2.3504023b0       20       1.2162183b-24
   2.3504023b0       40       1.8367099b-40
```

Maxima by Example:
Ch.10: Fourier Series, Fourier and Laplace Transforms *

Edwin L. Woollett

September 16, 2010

Contents

10.1 Introduction 3
10.2 Fourier Series Expansion of a Function 3
10.2.1 Fourier Series Expansion of a Function over $(-\pi, \pi)$ 3
10.2.2 Fourier Series Expansion of $f(x) = x$ over $(-\pi, \pi)$ 4
10.2.3 The **calculus/fourie.mac** Package: **fourier**, **foursimp**, **fourexpand** 5
10.2.4 Fourier Series Expansion of a Function Over $(-p, p)$ 7
10.2.5 Fourier Series Expansion of the Function $|x|$ 8
10.2.6 Fourier Series Expansion of a Rectangular Pulse 11
10.2.7 Fourier Series Expansion of a Two Element Pulse 13
10.2.8 Exponential Form of a Fourier Series Expansion 16
10.3 Fourier Integral Transform Pairs 18
10.3.1 Fourier Cosine Integrals and **fourintcos(..)** 18
10.3.2 Fourier Sine Integrals and **fourintsin(..)** 19
10.3.3 Exponential Fourier Integrals and **fourint** 21
10.3.4 Example 1: Even Function 21
10.3.5 Example 2: Odd Function 24
10.3.6 Example 3: A Function Which is Neither Even nor Odd 26
10.3.7 Dirac Delta Function $\delta(x)$ 28
10.3.8 Laplace Transform of the Delta Function Using a Limit Method 31
10.4 Laplace Transform Integrals 32
10.4.1 Laplace Transform Integrals: **laplace(..)**, **specint(..)** 32
10.4.2 Comparison of **laplace** and **specint** 32
10.4.3 Use of the Dirac Delta Function (Unit Impulse Function) **delta** with **laplace(..)** 36
10.5 The Inverse Laplace Transform and Residues at Poles 36
10.5.1 **ilt**: Inverse Laplace Transform 36
10.5.2 **residue**: Residues at Poles 37

---

*This version uses **Maxima 5.18.1**. This is a live document. Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions to `woollett@charter.net`

COPYING AND DISTRIBUTION POLICY
This document is part of a series of notes titled
"Maxima by Example" and is made available
via the author's webpage http://www.csulb.edu/~woollett/
to aid new users of the Maxima computer algebra system.

NON-PROFIT PRINTING AND DISTRIBUTION IS PERMITTED.

You may make copies of this document and distribute them
to others as long as you charge no more than the costs of printing.

These notes (with some modifications) will be published in book form
eventually via Lulu.com in an arrangement which will continue
to allow unlimited free download of the pdf files as well as the option
of ordering a low cost paperbound version of these notes.

Feedback from readers is the best way for this series of notes to become more helpful to new users of Maxima.
All comments and suggestions for improvements will be appreciated and carefully considered.

**The Maxima code examples in this chapter were generated using the XMaxima graphics interface on a Windows XP computer, and copied into a fancy verbatim environment in a latex file which uses the fancyvrb and color packages.
We use qdraw.mac for plots (see Ch.5), which uses draw2d defined in share/draw/draw.lisp.**

Maxima.sourceforge.net. Maxima, a Computer Algebra System. Version 5.18.1
(2009). http://maxima.sourceforge.net/

10.1 Introduction

In chapter 10 we discuss the Fourier series expansion of a given function, the computation of Fourier transform integrals, and the calculation of Laplace transforms (and inverse Laplace transforms).

10.2 Fourier Series Expansion of a Function

10.2.1 Fourier Series Expansion of a Function over $(-\pi,\ \pi)$

A Fourier series expansion designed to represent a given function **f(x)** defined over a finite interval $(-\pi, \pi)$, is a sum of terms

$$f(x) = \frac{1}{2}\, a_0 + \sum_{n=1}^{\infty} [a_n \cos(n\, x) + b_n \sin(n\, x)] \tag{10.1}$$

and the constant coefficients $(a_n,\ b_n)$ are

$$a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(y) \cos(n\, y)\, dy, \quad b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(y) \sin(n\, y)\, dy. \tag{10.2}$$

(For a derivation of these equations see Sec.10.2.8 and Eqs.(10.20) and (10.21).)

Whether or not you are working with a function which is periodic, the Fourier expansion will represent a periodic function for all **x**, in this case having period $2\,\pi$.

The first term of the expansion

$$\frac{1}{2}\, a_0 = \frac{1}{2\,\pi} \int_{-\pi}^{\pi} f(y)\, dy = \langle f(x) \rangle \tag{10.3}$$

is the (un-weighted) average of **f(x)** over the domain $(-\pi, \pi)$. Hence a_0 will always be twice the average value of the function over the domain.

If **f(x)** is an even function ($f(-x) = f(x)$), then only the $\cos(n\, x)$ terms contribute. If **f(x)** is an odd function ($f(-x) = -f(x)$), then only the $\sin(n\, x)$ terms contribute.

If you are trying to find a fourier expansion for a function or expression **f(x)** which is a homemade function which involves "if..then..else" constructs, it is necessary to do the preliminary work "by hand".

On the other hand, if the given function is a smooth function defined in terms of elementary functions and polynomials, or includes **abs** of elements of the function, one can use the **fourie** package to do most of the work in obtaining the desired Fourier series expansion. This package is **`calculus/fourie.mac`**, and there is also a short **demo** file **`fourie.dem`**.

Although many secondary functions defined in this mac file are usable once you load the package, they are not documented in the Maxima manual. The Maxima manual gives a brief definition of the primary tools available, but gives no examples of use, nor is there an **example** file for such primary functions as **fourier**, **foursimp**, and **fourexpand**.

If the Fourier series integrals needed for the coefficients are too difficult for **integrate**, you should use the Quadpack function **`quad_qawo`** described in Chapter 8, Numerical Integration.

10.2.2 Fourier Series Expansion of $f(x) = x$ over $(-\pi, \pi)$

We first use the coefficient formulas Eq.(10.2) to find the Fourier expansions of this simple linear function $f(x) = x$ by hand.

Because the average value of **f(x)** over the domain $(-\pi, \pi)$ is zero, $a_0 = 0$. Because **f(x)** is an odd function, we have $a_n = 0$ for all $n > 0$.

```
(%i1) (declare(n,integer), assume(n > 0 ), facts() );
(%o1)                          [kind(n, integer), n > 0]
(%i2) define(b(n),integrate(x*sin(n*x),x,-%pi,%pi)/%pi );
                                                 n
                                         2 (- 1)
(%o2)                          b(n) := - --------
                                             n
(%i3) map('b,makelist(i,i,1,7));
                                    2    1  2    1  2
(%o3)                        [2, - 1, -, - -, -, - -, -]
                                    3    2  5    3  7
(%i4) fs(nmax) := sum( b(m)*sin(m*x),m,1,nmax)$
(%i5) map('fs,[1,2,3,4] );
                                           2 sin(3 x)
(%o5) [2 sin(x), 2 sin(x) - sin(2 x), ---------- - sin(2 x) + 2 sin(x),
                                               3
                                       sin(4 x)   2 sin(3 x)
                                     - -------- + ---------- - sin(2 x) + 2 sin(x)]
                                          2            3
```

The list contains, in order, the lowest approximation $2\sin(x)$ which retains only the $n = 1$ term in the expansion, the two term approximation $2\sin(x) - \sin(2x)$, which includes the $n = 1, 2$ terms, etc. We now load the **draw** package and the **qdraw** package (the latter available with Ch. 5 material on the author's webpage) to make two simple plots. We first make a plot showing the function $f(x) = x$ in blue, the one term approximation (`fs(1) = 2 sin(x)`) in red, and the two term approximation (`fs(2) = 2 sin(x) - sin(2 x)`) in green.

```
(%i6) (load(draw), load(qdraw) )$
               qdraw(...), qdensity(...), syntax: type qdraw();

(%i7) qdraw( xr(-5.6,5.6),yr(-4,4),
               ex([x,fs(1),fs(2)],x,-%pi,%pi),key(bottom) )$
```

In this plot (see next page), we have taken control of the **x** and **y** range, using the approximate fudge factor `1.4` to relate the horizontal canvas extent to the vertical canvas extent (see our discussion in Ch. 5 if this is all new to you) to get the geometry approximately correct. In the next plot we include one, two, three and four term approximations.

```
(%i8) qdraw( xr(-5.6,5.6),yr(-4,4),
               ex([x,fs(1),fs(2),fs(3),fs(4)],x,-%pi,%pi),key(bottom) )$
```

with the four term approximation in purple being a closer approximation to $f(x) = x$ (see the figure on the next page).

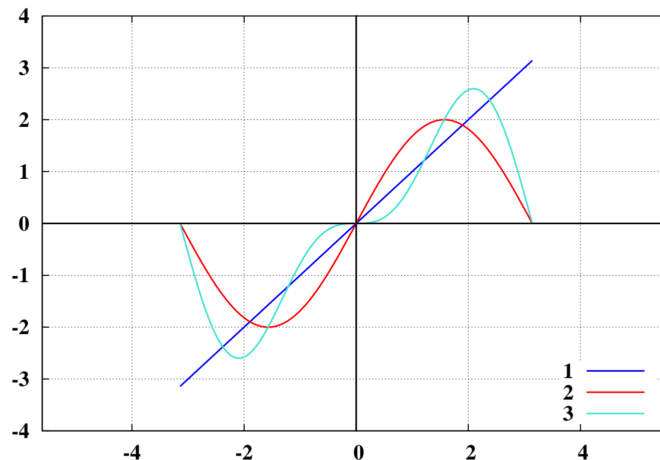

Figure 1: One and Two Term Approximations

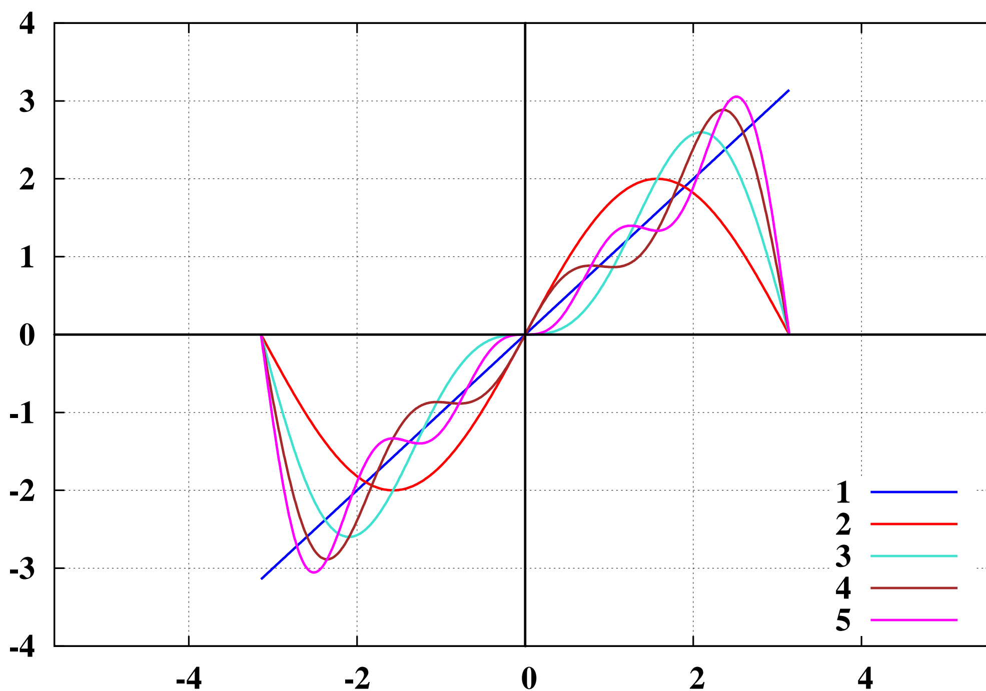

Figure 2: 1, 2, 3, and 4 Term Approximations

10.2.3 The calculus/fourie.mac Package: fourier, foursimp, fourexpand

Now let's show how to get the expansions of this simple linear function using the package **calculus/fourie.mac**. A curious feature of **fourie.mac** is the role that the symbol `n` plays in the calculation of the fourier coefficients. If you look at the code of **calculus/fourie.mac** (about a four page text file), you will see that the function **fourier** calls either **fourcos**, **foursin**, or **fourcoeff**, and each of these latter functions declare `n` to be a local variable, and then have the statement (inside the **block**) `assume( n > 0 )`, which "leaks out" of the **block** to become a global assumption. These functions then call a package function **adefint(..)** to integrate the function to be expanded times either $\cos(n\,\pi\,x/p)$ or $\sin(n\,\pi\,x/p)$, (where p will be π for our expansion domain here).

Finally, after dealing with possible instances of **abs(..)** in the function to be expanded, **adefint** calls either **ldefint** or **integrate** with an argument involving these same **n** dependent trig functions.

Here is a small test of that leakage for both an **assume** statement and a **declare** statement.

```
(%i1) facts();
(%o1)                               []
(%i2) f(m) := block([n],declare(n,integer),assume( n > 0 ),
                 if m < 2 then n :2 else n:3,(2*n*m) )$
(%i3) f(1);
(%o3)                               4
(%i4) facts();
(%o4)                   [kind(n, integer), n > 0]
(%i5) is(n>0);
(%o5)                              true
```

Another curious feature of the **fourie.mac** package is that **n** is not declared to be an integer, and unless the user of the package does this first, the integration routine may ask questions about signs which may seem irrelevant to the result. We avoid that trap here and use **declare(n, integer)** before calling **fourier**.

```
(%i1) facts();
(%o1)                               []
(%i2) (load(fourie), facts() );
(%o2)                               []
(%i3) (declare(n,integer), facts() );
(%o3)                      [kind(n, integer)]
(%i4) clist : fourier(x,x,%pi);
(%t4)                             a  = 0
                                   0

(%t5)                             a  = 0
                                   n


                                             n
                                     2 (- 1)
(%t6)                        b  = - --------
                              n         n

(%o6)                        [%t4, %t5, %t6]
(%i7) facts();
(%o7)                   [kind(n, integer), n > 0]
(%i8) fs(nmax) := fourexpand(clist,x,%pi, nmax )$
(%i9) map( 'fs, [1,2,3,4] );
                                         2 sin(3 x)
(%o9) [2 sin(x), 2 sin(x) - sin(2 x), ---------- - sin(2 x) + 2 sin(x),
                                             3
                                 sin(4 x)   2 sin(3 x)
                               - -------- + ---------- - sin(2 x) + 2 sin(x)]
                                    2           3
(%i10) b(n);
(%o10)                             b(n)
```

Some comments: you can use **foursimp** to get extra simplification of the fourier coefficients (if needed) before defining what I call **clist** (for coefficient list) which is the list **fourexpand** needs to generate the expansions you want. Note **fourier** produces a separate output for a_0 and a_n, with the second meant for $n = 1, 2, \ldots$, and there is never an output for b_0. We have defined the small function **fs(nmax)** to make it easier to call

fourexpand with any value of $\mathbf{nmax}$ and to allow the function to be mapped onto a list of integers to show us the first few approximations. Note that once you call **fourier**, the assumption $\mathbf{n > 0}$ becomes a global fact. Also note that the **fourie.mac** package does not define a Maxima function $\mathbf{b(n)}$, although you could use:

```
(%i11) define(b(n),rhs(%t6) );
                                                     n
                                               2 (- 1)
(%o11)                              b(n) := - --------
                                                   n
(%i12) map( 'b, makelist(i,i,1,7) );
                                          2    1  2    1  2
(%o12)                           [2, - 1, -, - -, -, - -, -]
                                          3    2  5    3  7
```

If you look at the Maxima code in `fourie.mac`, you see that because we have an "odd" function $\mathbf{f(x) = x}$, **fourier** calls the package function **foursin**, which calls package function **adefint**, which calls the core Maxima function **ldefint** for this case.

Those expansion expressions can then be used for plots as above.

10.2.4 Fourier Series Expansion of a Function Over $(-\mathbf{p},\ \mathbf{p})$

The Fourier series expansion of a function defined over the interval $-\mathbf{p} \le \mathbf{x} \le \mathbf{p}$ (and whose Fourier expansion will represent a function which has a period $\mathbf{2\,p}$) can be found from the expansion over the interval $(-\pi, \pi)$ which we have been using above by a simple change of variables in the integrals which appear in Eqs.(10.2) and (10.1).

However, we will simply use the results derived in Sec.10.2.8, and written down in Eqns (10.18) and (10.19).

$$\mathbf{f(x) = \frac{1}{2}\,a_0 + \sum_{n=1}^{\infty}\left[a_n \cos\left(\frac{\pi\,n\,x}{p}\right) + b_n \sin\left(\frac{\pi\,n\,x}{p}\right)\right]} \tag{10.4}$$

with the corresponding coefficients (for $\mathbf{a_n}$, $\mathbf{n = 0, 1, ...}$, for $\mathbf{b_n}$, $\mathbf{n = 1, ...}$):

$$\mathbf{a_n = \frac{1}{p}\int_{-p}^{p} f(y)\cos\left(\frac{\pi\,n\,y}{p}\right)dy, \quad b_n = \frac{1}{p}\int_{-p}^{p} f(y)\sin\left(\frac{\pi\,n\,y}{p}\right)dy.} \tag{10.5}$$

We need to warn the user of the package **fourie.mac** that they use the symbol $\mathbf{a_0}$ to mean *our* $\mathbf{a_0/2}$.

Our $\mathbf{a_0}$ is defined by

$$\mathbf{a_0 = \frac{1}{p}\int_{-p}^{p} f(x)\,dx} \tag{10.6}$$

The **fourie.mac** package's definition of *their* $\mathbf{a_0}$ is

$$\mathbf{[a_0]_{fourie.mac} = \frac{1}{2\,p}\int_{-p}^{p} f(x)\,dx} \tag{10.7}$$

which defines the average value of the function over the domain and which becomes the first term of the fourier series expansion they provide.

10.2.5 Fourier Series Expansion of the Function $|x|$

We define the function to have period 4 with $f(x) = |x|$ for $-2 \le x \le 2$. This function is an even function of $\mathbf{x}$ so the $\sin$ coefficients b_n are all zero. The average value of $|x|$ over the domain $(-2, 2)$ is greater than zero so we willl have a non-zero coefficient a_0 which we will calculate separately. We do this calculation by hand here.

Note that the Maxima function **integrate** cannot cope with **abs(x)**:

```
(%i13) integrate(abs(x)*cos(n*%pi*x/2),x,-2,2)/2;
                              2
                             /
                             [                %pi n x
                             I     abs(x)  cos(-------) dx
                             ]                   2
                             /
                              - 2
(%o13)                       ---------------------------
                                          2
```

so we will split up the region of integration into two sub-intervals: $-2 \le x \le 0$, in which $|x| = -x$, and the interval $0 \le x \le 2$ in which $|x| = x$. We will use the formula Eq. (10.5) for the coefficients a_n (note that $a_n = 0$ if $\mathbf{n}$ is even) and the expansion formula Eq. (10.4).

```
(%i1) (declare(n,integer),assume(n > 0), facts() );
(%o1)                         [kind(n, integer), n > 0]
(%i2) a0 :integrate(-x,x,-2,0)/2 + integrate(x,x,0,2)/2;
(%o2)                                     2
(%i3) an : integrate((-x)*cos(n*%pi*x/2),x,-2,0)/2 +
           integrate(x*cos(n*%pi*x/2),x,0,2)/2;
                                        n
                                 4 (- 1)        4
(%o3)                            -------- - -------
                                    2   2        2   2
                                 %pi   n     %pi   n
(%i4) an : (ratsimp(an), factor(%%) );
                                           n
                                   4 ((- 1)  - 1)
(%o4)                              --------------
                                         2   2
                                      %pi   n
(%i5) define(a(n),an);
                                               n
                                       4 ((- 1)  - 1)
(%o5)                          a(n) := --------------
                                             2   2
                                          %pi   n
(%i6) map( 'a, [1,2,3,4,5] );
                              8           8              8
(%o6)                     [- ----, 0, - ------, 0, - -------]
                                2           2              2
                             %pi        9 %pi         25 %pi
(%i7) fs(nmax) := a0/2 + sum(a(m)*cos(m*%pi*x/2),m,1,nmax)$
```

```
(%i8) map('fs, [1, 3, 5] );
                   %pi x             3 %pi x           %pi x
           8 cos(-----)        8 cos(-------)    8 cos(-----)
                     2                  2                2
(%o8) [1 - ------------, - -------------- - ------------ + 1,
                  2                  2                2
                %pi             9 %pi              %pi
                                       5 %pi x           3 %pi x            %pi x
                                 8 cos(-------)    8 cos(-------)    8 cos(-----)
                                         2                 2                 2
                               - -------------- - -------------- - ------------ + 1]
                                         2                 2               2
                                    25 %pi              9 %pi            %pi
(%i9) (load(draw),load(qdraw))$
              qdraw(...), qdensity(...), syntax: type qdraw();

(%i10) qdraw( ex([abs(x),fs(1)],x,-2,2),key(bottom) )$
(%i11) qdraw( ex([abs(x),fs(1),fs(3)],x,-2,2),key(bottom) )$
(%i12) qdraw( ex([abs(x),fs(5) ],x,-2,2),key(bottom) )$
```

The first function in the plot list is $|x|$, represented by **abs(x)**, which appears in the color blue. We see that the expansion out to $n = 5$ provides a close fit to $|x|$. Here is that comparison:

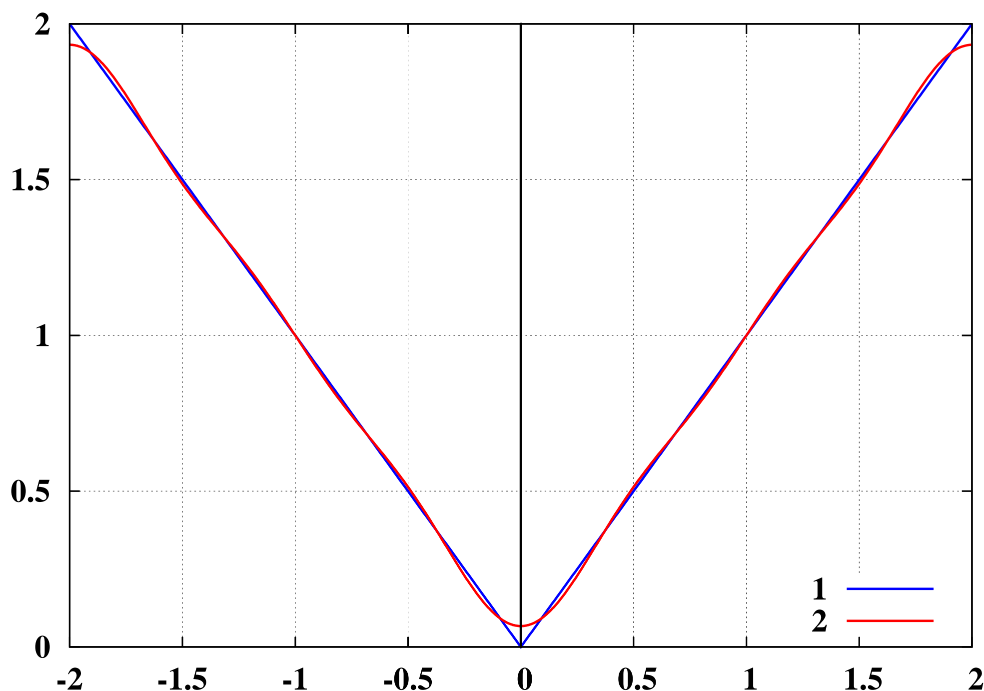

Figure 3: $n = 5$ Approximation to $|x|$

We now try out the package **fourie.mac** on this example:

```
(%i1) ( load(fourie),facts() );
(%o1)                                  []
(%i2) (declare(n,integer),facts());
(%o2)                          [kind(n, integer)]
```

```
(%i3) fourier(abs(x),x,2);
(%t3)                              a  = 1
                                    0

                                       n
                              4 (- 1)        4
(%t4)                      a  = -------- - -------
                            n      2  2        2  2
                              %pi  n     %pi  n

(%t5)                              b  = 0
                                    n

(%o5)                           [%t3, %t4, %t5]
(%i6) clist : foursimp(%);
(%t6)                              a  = 1
                                    0

                                          n
                                4 ((- 1)  - 1)
(%t7)                        a  = --------------
                              n          2  2
                                    %pi  n

(%t8)                              b  = 0
                                    n

(%o8)                           [%t6, %t7, %t8]
(%i9) facts();
(%o9)                    [kind(n, integer), n > 0]
(%i10) fs(nmax) := fourexpand(clist,x,2,nmax )$
(%i11) map( 'fs, [1,3,5] );
                 %pi x           3 %pi x          %pi x
            8 cos(-----)    8 cos(-------)   8 cos(-----)
                   2                2               2
(%o11) [1 - ------------, - -------------- - ------------ + 1,
                  2                2              2
                %pi            9 %pi           %pi
                                  5 %pi x          3 %pi x          %pi x
                            8 cos(-------)   8 cos(-------)   8 cos(-----)
                                    2                2               2
                          - -------------- - -------------- - ------------ + 1]
                                    2                2              2
                              25 %pi            9 %pi          %pi
```

Notice that $(a_0)_{\mathbf{fourie}} = 1 = \frac{1}{2}\,(a_0)_{\mathbf{ourdef}}$, (see Eq. (10.7)) so that **`fourie.mac`**'s expansion starts off with the term a_0 rather than $\frac{1}{2}\,a_0$. Of course, the actual end results look the same, with the first term in this example being $\mathbf{1}$, which is the average value of $|x|$ over the domain $(-2, 2)$.

Here we chose to use the package function **foursimp** to simplify the appearance of the coefficients. We see that **`fourie.mac`** is able to cope with the appearance of **abs(x)** in the integrand, and produces the same coefficients and expansions as were found ”by hand”.

10.2.6 Fourier Series Expansion of a Rectangular Pulse

We define **f(x)** to be a function of period 4, with $f = 0$ for $-2 \le x < -1$, $3/2$ for $-1 \le x \le 1$ and $f = 0$ for $1 < x \le 2$.

```
(%i1) f(x):= if x >= -1 and x <= 1 then 3/2 else 0$
(%i2) map('f, [-3/2,-1,0,1,3/2] );
                                      3  3  3
(%o2)                            [0, -, -, -, 0]
                                      2  2  2
(%i3) (load(draw),load(qdraw) )$
               qdraw(...), qdensity(...), syntax: type qdraw();

(%i4) qdraw( yr(-0.5,2), ex1(f(x),x,-2,2,lw(5),lc(blue) ) )$
```

The plot of **f(x)** shows a square pulse with height $3/2$ above the **x** axis and with a width of 2 units over the interval $-1 \le x \le 1$. Over the rest of the domain $-2 \le x \le 2$, **f(x)** is defined to be zero.

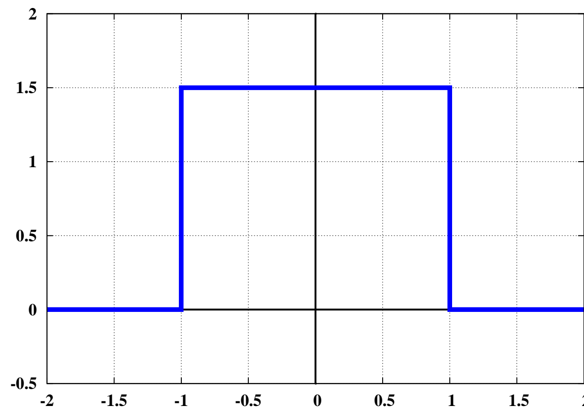

Figure 4: Rectangular Pulse of Height $3/2$

Although we can make a plot of the function **f(x)** as defined , Maxima's **integrate** function cannot do anything useful with it, and hence neither can **fourie.mac** at the time of writing.

```
(%i5) integrate(f(x),x,-2,2);
               2
              /
              [                                        3
(%o5)         I    (if (x >= - 1) and (x <= 1) then - else 0) dx
              ]                                        2
              /
               - 2
```

Hence we must compute the Fourier series expansion "by hand". We see that **f(x)** is an even function $(f(-x) = f(x))$, so only the $\cos(n\,\pi\,x/2)$ terms will contribute, so we only need to calculate the a_n coefficients.

```
(%i6) (declare(n,integer), assume(n>0),facts() );
(%o6)                       [kind(n, integer), n > 0]
```

```
(%i7) a0 : (1/2)*integrate( (3/2),x,-1,1 );
                                        3
(%o7)                                   -
                                        2
(%i8) define(a(n),(1/2)*integrate((3/2)*cos(n*%pi*x/2),x,-1,1));
                                             %pi n
                                       3 sin(-----)
                                               2
(%o8)                          a(n) := ------------
                                          %pi n
(%i9) map( 'a, makelist(i,i,1,7) );
                         3         1        3           3
(%o9)                  [---, 0, - ---, 0, -----, 0, - -----]
                        %pi       %pi     5 %pi       7 %pi
```

We see that for $n > 0$, $a_n = 0$ for n even, and the non-zero coefficients have $n = 1, 3, 5, 7$, ... Hence we only get a better approximation if we increase n_{max} by 2 each time.

```
(%i10) fs(nmax) := a0/2 + sum( a(m)*cos(m*%pi*x/2),m,1,nmax )$
(%i11) map( 'fs, [1,3] );
                   %pi x             3 %pi x          %pi x
              3 cos(-----)        cos(-------)   3 cos(-----)
                      2      3           2                2      3
(%o11)        [------------ + -, - ------------ + ------------ + -]
                  %pi        4         %pi            %pi         4
(%i12) qdraw( yr(-0.5,2),ex([f(x),fs(1),fs(3) ],x,-2,2) )$
(%i13) qdraw( yr(-0.5,2),ex([f(x),fs(11) ],x,-2,2) )$
```

The plot with the approximations `fs(1)` and `fs(3)` was drawn first.

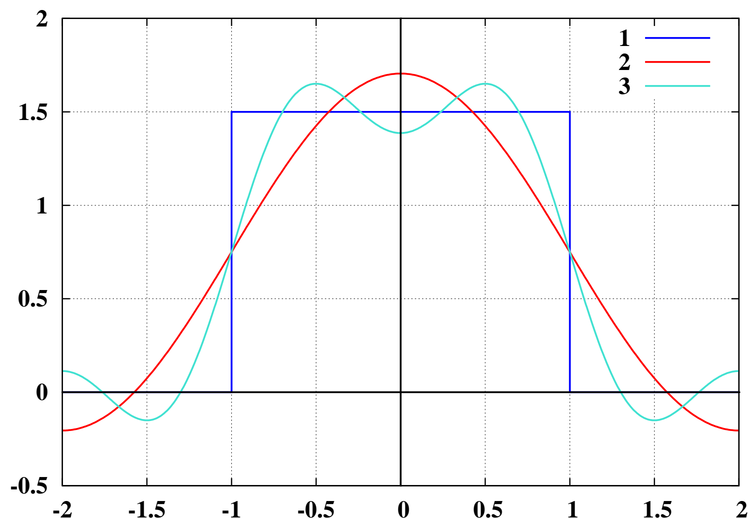

Figure 5: $nmax = 1, 3$ Approx. to Rectangular Pulse

Then a plot showing the **fs(11)** approximation:

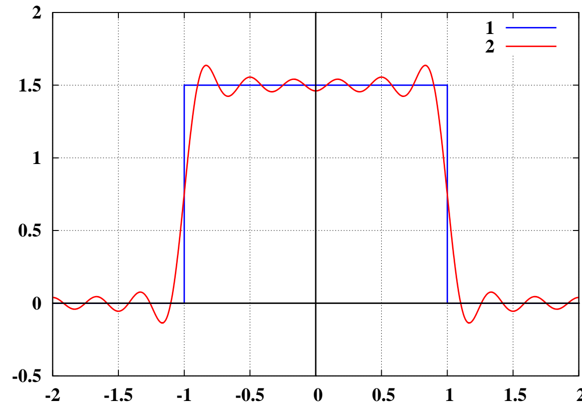

Figure 6: $\mathbf{nmax} = \mathbf{11}$ Approx. to Rectangular Pulse

10.2.7 Fourier Series Expansion of a Two Element Pulse

We compute the Fourier series expansion of a function which has the definition over the interval $(-10, 10)$ (and whose Fourier series expansion will be a function with period $\mathbf{20}$ for all $\mathbf{x}$) given by: for $-10 \leq x < -5$, $f = 0$, and for $-5 \leq x < 0$, $f = -5$, and for $0 \leq x \leq 5$, $f = 5$, and for $5 < x \leq 10$, $f = 0$. First we define such a function for our plots.

```
(%i1) f(x):= if x >= -5 and x < 0 then -5
             elseif x >= 0 and x <= 5 then 5 else 0$
(%i2) map('f,[-6,-5,-1,0,1,5,6]);
(%o2)                        [0, - 5, - 5, 5, 5, 5, 0]
```

and plot the function

```
(%i3) ( load(draw),load(qdraw) )$
              qdraw(...), qdensity(...), syntax: type qdraw();
(%i4) qdraw( yr(-8,8), ex1(f(x),x,-10,10,lw(5),lc(blue) ) )$
```

which looks like

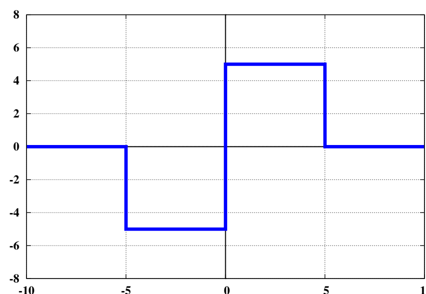

Figure 7: Two Element Pulse with Period $\mathbf{20}$

Since we have an odd function, only terms like $\mathbf{b_n \sin(n\, \pi\, x/10)}$ will contribute to the expansion, but, just for practice, we find $\mathbf{a_n}$ too.

```
(%i5) a0 : (1/10)*( integrate( -5, x, -5, 0 ) +
                              integrate( 5, x, 0, 5 )  );
(%o5)                                    0
(%i6) an : (1/10)*(integrate( -5*cos( n*%pi*x/10 ), x, -5, 0 ) +
                       integrate(5*cos(n*%pi*x/10), x, 0, 5 )  );
(%o6)                                    0
(%i7) bn : ( (1/10)*(integrate( -5*sin(n*%pi*x/10), x, -5, 0 ) +
             integrate( 5*sin(n*%pi*x/10), x, 0, 5 )  ),
              ratsimp(%%) );
                                       %pi n
                                10 cos(-----) - 10
                                         2
(%o7)                         - ------------------
                                      %pi n
(%i8) define( b(n), bn );
                                           %pi n
                                    10 cos(-----) - 10
                                             2
(%o8)                    b(n) := - ------------------
                                          %pi n
(%i9) map('b,makelist(i,i,1,7));
                     10    10     10         2     10      10
(%o9)              [---, ---, -----, 0, ---, -----, -----]
                    %pi   %pi   3 %pi      %pi   3 %pi   7 %pi
(%i10) fs(nmax) :=  sum( b(m)*sin(m*%pi*x/10), m, 1, nmax )$
(%i11) map('fs,[1,2,3]);
                %pi x           %pi x            %pi x
       10 sin(-----)  10 sin(-----)   10 sin(-----)
                 10              5               10
(%o11) [-------------, ------------- + -------------,
             %pi            %pi              %pi
                                    3 %pi x          %pi x           %pi x
                             10 sin(-------)   10 sin(-----)   10 sin(-----)
                                      10               5               10
                             --------------- + ------------- + -------------]
                                  3 %pi             %pi             %pi
(%i12) qdraw(  xr(-15, 15), yr(-10, 10),
                         ex( [f(x), fs(1), fs(2) ], x, -10, 10 ) )$
```

The plot of **f(x)** with the two lowest approximations looks like

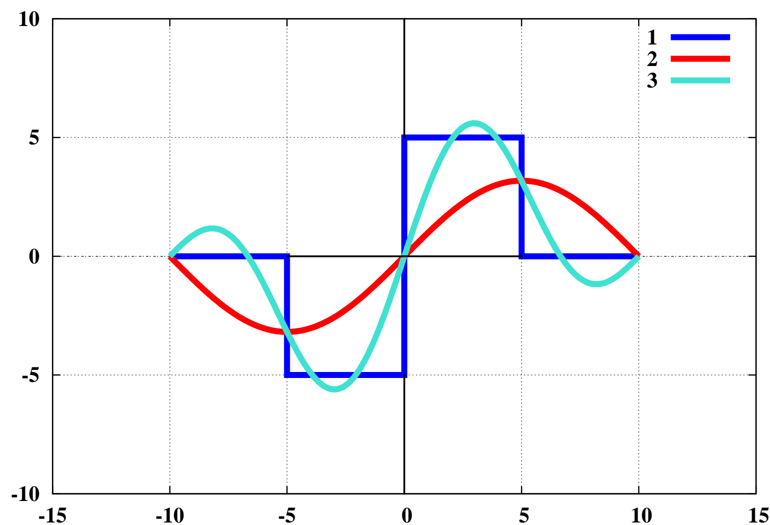

Figure 8: Two Lowest Order Approximations

The expansion **fs(11)** does pretty well as a rough approximation

```
(%i13) qdraw( xr(-15, 15), yr(-10, 10),
                    ex(   [ f(x), fs(11) ], x, -10, 10 ) )$
```

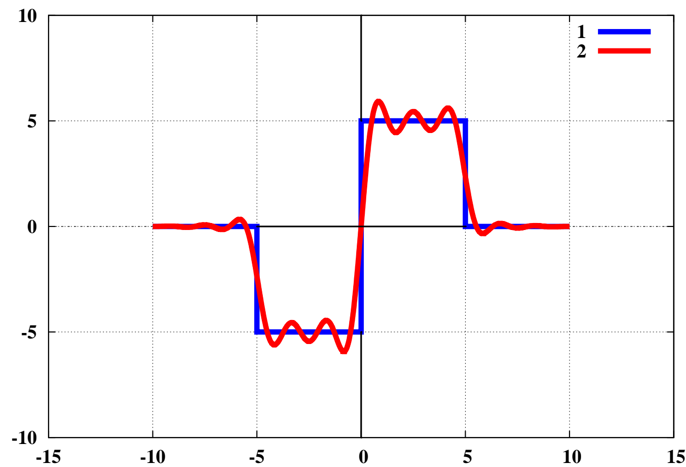

Figure 9: $\mathrm{nmax} = 11$ Approx. to Two Element Pulse

Whether you are working with a periodic function or not, the Fourier series expansion always represents a periodic function for all **x**. For this example, the period is **20**, and if we make a plot of **fs(11)** over the range `[-10, 30]`, we are including two periods.

```
(%i14) qdraw(yr(-10, 10),ex( fs(11), x, -10 , 30 ) )$
```

which looks like

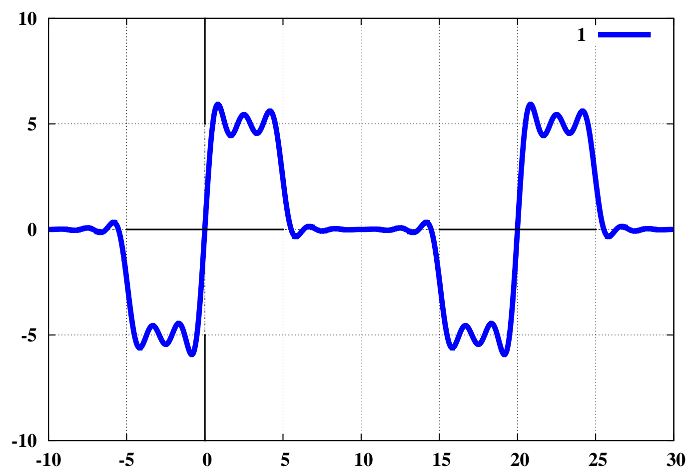

Figure 10: $\mathrm{nmax} = 11$ Approx. Drawn for Two Periods

10.2.8 Exponential Form of a Fourier Series Expansion

The exponential form of a Fourier series can be based on the completeness and orthogonality (in the Hermitian sense) of the set of exponential functions

$$\left\{\exp\left(\frac{2\,\pi\,i\,n\,x}{b-a}\right)\right\} \qquad (n = 0,\ \pm 1,\ \pm 2,\ \cdots) \tag{10.8}$$

on the interval $(a,\ b)$. Define the function $\phi_n(x)$ as

$$\phi_n(x) = \exp\left(\frac{-2\,\pi\,i\,n\,x}{b-a}\right). \tag{10.9}$$

This function has the properties

$$\phi_{-n}(x) = \phi_n(x)^*, \qquad \phi_n(x)\,\phi_n(x)^* = 1. \tag{10.10}$$

in which the asterisk indicates the complex conjugate.

The (Hermitian) orthogonality of the $\{\phi_n(x)\}$ over $(a,\ b)$ is expressed by

$$\int_a^b \phi_n(x)\,\phi_m(x)^*\,dx = (b-a)\,\delta_{n\,m}, \tag{10.11}$$

in which the Kronecker delta symbol is

$$\delta_{n\,m} = \left\{\begin{array}{lll} 1 & \text{for} & n = m \\ 0 & \text{for} & n \neq m. \end{array}\right\} \tag{10.12}$$

Eq.(10.11) is clearly true for $n = m$, using Eq.(10.10). For $m \neq n$, we can simplify this integral by using the exponential product law $e^A\,e^B = e^{A+B}$, letting $r = m - n \neq 0$, and changing the variable of integration $x \to y$ via $x = (b-a)\,y + a$. The resulting integral in terms of y will then be over the interval $(0, 1)$. The differential $dx \to (b-a)\,dy$ and $(b-a)$ comes outside the integral. Also, inside the exponential, $x/(b-a) \to a/(b-a) + y$, and we can take outside an exponential function of a constant. Thus the integral is proportional to $\int_0^1 \exp(2\,\pi\,i\,r\,y)\,dy$, which is proportional to $\exp(2\,\pi\,i\,r) - 1 = \exp(2\,\pi\,i)^r - 1 = 0$.

We now write some function of x as a linear combination of the $\phi_n(x)$ with coefficients C_n to be determined.

$$f(x) = \sum_{n=-\infty}^{\infty} C_n\,\phi_n(x). \tag{10.13}$$

To find the $\{C_n\}$, we multiply both sides of Eq.(10.13) by $\phi_m(x)^*\,dx$, integrate both sides over the interval $(a,\ b)$, and use orthogonality, Eq.(10.11).

$$\begin{aligned}
\int_a^b f(x)\,\phi_m(x)^*\,dx &= \sum_{n=-\infty}^{\infty} C_n \int_a^b \phi_n(x)\,\phi_m(x)^*\,dx \\
&= \sum_{n=-\infty}^{\infty} C_n\,(b-a)\,\delta_{n\,m} \\
&= (b-a)\,C_m.
\end{aligned}$$

Hence we have for the coefficients

$$\mathbf{C_n} = \frac{1}{(\mathbf{b}-\mathbf{a})} \int_{\mathbf{a}}^{\mathbf{b}} \mathbf{f}(\mathbf{x})\, \phi_{\mathbf{n}}(\mathbf{x})^* \, \mathbf{dx}. \tag{10.14}$$

Inserting these coefficients into Eq.(10.13) we can write

$$\begin{aligned}
\mathbf{f}(\mathbf{x}) &= \sum_{\mathbf{n}=-\infty}^{\infty} \mathbf{C_n}\, \phi_{\mathbf{n}}(\mathbf{x}) \\
&= \sum_{-\infty}^{\infty} \exp\left(\frac{-2\,\pi\,\mathbf{i}\,\mathbf{n}\,\mathbf{x}}{\mathbf{b}-\mathbf{a}}\right) \frac{1}{\mathbf{b}-\mathbf{a}} \int_{\mathbf{a}}^{\mathbf{b}} \mathbf{f}(\mathbf{y}) \exp\left(\frac{2\,\pi\,\mathbf{i}\,\mathbf{n}\,\mathbf{y}}{\mathbf{b}-\mathbf{a}}\right) \mathbf{dy} \\
&= \frac{1}{\mathbf{b}-\mathbf{a}} \sum_{-\infty}^{\infty} \int_{\mathbf{a}}^{\mathbf{b}} \mathbf{f}(\mathbf{y}) \exp\left(\frac{2\,\pi\,\mathbf{i}\,\mathbf{n}\,(\mathbf{y}-\mathbf{x})}{\mathbf{b}-\mathbf{a}}\right) \mathbf{dy}.
\end{aligned}$$

We now separate out the $\mathbf{n} = \mathbf{0}$ term and combine the $\mathbf{n} = \pm\mathbf{m}$ terms into a sum over the positive integers.

$$\begin{aligned}
\mathbf{f}(\mathbf{x}) &= \frac{1}{\mathbf{b}-\mathbf{a}} \int_{\mathbf{a}}^{\mathbf{b}} \mathbf{f}(\mathbf{y})\, \mathbf{dy} + \frac{1}{\mathbf{b}-\mathbf{a}} \sum_{\mathbf{n}=1}^{\infty} \int_{\mathbf{a}}^{\mathbf{b}} \mathbf{f}(\mathbf{y}) \left[\exp\left(\frac{2\,\pi\,\mathbf{i}\,\mathbf{n}\,(\mathbf{y}-\mathbf{x})}{\mathbf{b}-\mathbf{a}}\right) + \exp\left(\frac{-2\,\pi\,\mathbf{i}\,\mathbf{n}\,(\mathbf{y}-\mathbf{x})}{\mathbf{b}-\mathbf{a}}\right)\right] \mathbf{dy} \\
&= \frac{1}{\mathbf{b}-\mathbf{a}} \int_{\mathbf{a}}^{\mathbf{b}} \mathbf{f}(\mathbf{y})\, \mathbf{dy} + \frac{2}{\mathbf{b}-\mathbf{a}} \sum_{\mathbf{n}=1}^{\infty} \int_{\mathbf{a}}^{\mathbf{b}} \mathbf{f}(\mathbf{y}) \cos\left(\frac{2\,\pi\,\mathbf{n}\,(\mathbf{y}-\mathbf{x})}{\mathbf{b}-\mathbf{a}}\right) \mathbf{dy}
\end{aligned}$$

Using the trig identity $\cos(\mathbf{A}-\mathbf{B}) = \cos\mathbf{A}\cos\mathbf{B} + \sin\mathbf{A}\sin\mathbf{B}$, we recover the trigonometric form of the Fourier series expansion of a function over the interval (a, b), which will represent a function which has period equal to (b - a) for all values of x.

$$\mathbf{f}(\mathbf{x}) = \frac{1}{2}\,\mathbf{a_0} + \sum_{\mathbf{n}=1}^{\infty} \left[\mathbf{a_n} \cos\left(\frac{2\,\pi\,\mathbf{n}\,\mathbf{x}}{\mathbf{b}-\mathbf{a}}\right) + \mathbf{b_n} \sin\left(\frac{2\,\pi\,\mathbf{n}\,\mathbf{x}}{\mathbf{b}-\mathbf{a}}\right)\right] \tag{10.15}$$

The coefficients are given by the integrals

$$\mathbf{a_n} = \frac{2}{\mathbf{b}-\mathbf{a}} \int_{\mathbf{a}}^{\mathbf{b}} \mathbf{f}(\mathbf{y}) \cos\left(\frac{2\,\pi\,\mathbf{n}\,\mathbf{y}}{\mathbf{b}-\mathbf{a}}\right) \mathbf{dy}, \quad \mathbf{b_n} = \frac{2}{\mathbf{b}-\mathbf{a}} \int_{\mathbf{a}}^{\mathbf{b}} \mathbf{f}(\mathbf{y}) \sin\left(\frac{2\,\pi\,\mathbf{n}\,\mathbf{y}}{\mathbf{b}-\mathbf{a}}\right) \mathbf{dy}. \tag{10.16}$$

The expansion starts with the term

$$\mathbf{f}(\mathbf{x}) = \frac{1}{2}\,\mathbf{a_0} + \cdots = \frac{1}{\mathbf{b}-\mathbf{a}} \int_{\mathbf{a}}^{\mathbf{b}} \mathbf{f}(\mathbf{y}) + \cdots \tag{10.17}$$

which is the (unweighted) average of $\mathbf{f}(\mathbf{x})$ over (a, b).

If we specialize to a function defined over the interval **(-p, p)**, whose Fourier expansion will represent a function which has period **2 p** for all **x**, the above results have the replacements $2/(\mathbf{b}-\mathbf{a}) \to 2/(2\,\mathbf{p}) \to 1/\mathbf{p}$, and the appropriate exspansion equations are

$$\mathbf{f}(\mathbf{x}) = \frac{1}{2}\,\mathbf{a_0} + \sum_{\mathbf{n}=1}^{\infty} \left[\mathbf{a_n} \cos\left(\frac{\pi\,\mathbf{n}\,\mathbf{x}}{\mathbf{p}}\right) + \mathbf{b_n} \sin\left(\frac{\pi\,\mathbf{n}\,\mathbf{x}}{\mathbf{p}}\right)\right] \tag{10.18}$$

with coefficients

$$\mathbf{a_n} = \frac{1}{\mathbf{p}} \int_{-\mathbf{p}}^{\mathbf{p}} \mathbf{f}(\mathbf{y}) \cos\left(\frac{\pi\,\mathbf{n}\,\mathbf{y}}{\mathbf{p}}\right) \mathbf{dy}, \quad \mathbf{b_n} = \frac{1}{\mathbf{p}} \int_{-\mathbf{p}}^{\mathbf{p}} \mathbf{f}(\mathbf{y}) \sin\left(\frac{\pi\,\mathbf{n}\,\mathbf{y}}{\mathbf{p}}\right) \mathbf{dy}. \tag{10.19}$$

If we specialize further to $p = \pi$, the appropriate equations are

$$f(x) = \frac{1}{2}\,a_0 + \sum_{n=1}^{\infty}\left[a_n \cos(n\,x) + b_n \sin(n\,x)\right] \tag{10.20}$$

with coefficients

$$a_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(y)\cos(n\,y)\,dy, \quad b_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(y)\sin(n\,y)\,dy. \tag{10.21}$$

10.3 Fourier Integral Transform Pairs

10.3.1 Fourier Cosine Integrals and fourintcos(..)

Given some function $f(x)$ defined for $x \geq 0$, we define the Fourier cosine transform of this function as

$$F_C(f,\omega) = \frac{2}{\pi}\int_0^{\infty} \cos(\omega\,x)\,f(x)\,dx \tag{10.22}$$

The given function $f(x)$ can then be written as an integral over positive values of ω:

$$f(x) = \int_0^{\infty} F_C(f,\omega)\,\cos(\omega\,x)\,d\omega \tag{10.23}$$

The two equations (10.22) and (10.23) are an example of a ”Fourier transform pair”, which include conventions about where to place the factor of $2/\pi$.

Here is a simple example. We let the function be $f(x) = \sin(x)\,e^{-x}$, defined for $x \geq 0$. Let the letter **w** be used to stand for ω. We first calculate the Fourier cosine transform $F_C(f,\omega)$ (we use the symbol **fcw** in our work here) using **integrate** and then show that the inverse integral over ω gives back the original function.

```
(%i1) f:sin(x)*exp(-x)$
(%i2) integrate(f*cos(w*x),x,0,inf);
                                                    2
                                     2         2 w
(%o2)                              ------ - --------
                                     4           4
                                   w  + 4   2 w  + 8
(%i3) fcw : (2/%pi)*ratsimp(%);
                                              2
                                       2 (w  - 2)
(%o3)                               - -------------
                                              4
                                       %pi (w  + 4)
(%i4) integrate(fcw*cos(w*x),w,0,inf);
Is  x  positive, negative, or zero?
p;
                                          - x
(%o4)                                   %e    sin(x)
```

We can also use the package **fourie.mac**, which we explored in the section on Fourier series. This package provides for the calculation of our Fourier cosine transform integral via the **fourintcos(expr,var)** function. The function **fourintcos(f,x)** returns an answer with the label a_z which contains the Fourier cosine integral $F_C(f, z)$, with the letter **z** being the package convention for what we called **w** (ω),

```
(%i5) load(fourie);
(%o5) C:/PROGRA~1/MAXIMA~3.1/share/maxima/5.18.1/share/calculus/fourie.mac
(%i6) fourintcos(f,x);
                                                          2
                                         2               z
                                    2 (------ - ------)
                                         4          4
                                        z  + 4    z  + 4
(%t6)                        a  = -------------------
                              z             %pi

(%o6)                                    [%t6]
(%i7) az : ratsimp(rhs(%t6));
                                            2
                                        2 z  - 4
(%o7)                               - --------------
                                             4
                                      %pi z  + 4 %pi
(%i8) (2/%pi)*ratsimp(%pi*az/2);
                                            2
                                       2 (z  - 2)
(%o8)                                - ------------
                                              4
                                       %pi (z  + 4)
```

Thus **fourintcos(expr,var)** agrees with our definition of the Fourier cosine transform.

10.3.2 Fourier Sine Integrals and fourintsin(..)

Given some function $f(x)$ defined for $x \geq 0$, we define the Fourier sine transform of this function as

$$F_S(f, \omega) = \frac{2}{\pi} \int_0^{\infty} \sin(\omega\, x)\, f(x)\, dx \tag{10.24}$$

The given function $f(x)$ can then be written as an integral over positive values of ω:

$$f(x) = \int_0^{\infty} F_S(f, \omega)\, \sin(\omega\, x)\, d\omega \tag{10.25}$$

The two equations (10.24) and (10.25) are another ”Fourier transform pair”, which include conventions about where to place the factor of $2/\pi$.

Here is a simple example. We let the function be $f(x) = \cos(x)\,e^{-x}$, defined for $x \ge 0$. Let the letter **w** stand for ω. We first calculate the Fourier sine transform $F_S(f,\omega)$ (we use the symbol **fsw** in our work here) using **integrate** and then show that the inverse integral over ω gives back the original function.

```
(%i1) f:cos(x)*exp(-x)$
(%i2) assume(w>0)$
(%i3) integrate(f*sin(w*x),x,0,inf);
                                          3
                                         w
(%o3)                                 ------
                                        4
                                       w   + 4
(%i4) fsw : (2/%pi)*%;
                                           3
                                        2 w
(%o4)                              ------------
                                          4
                                  %pi  (w   + 4)
(%i5) integrate(fsw*sin(w*x),w,0,inf);
Is  x  positive, negative, or zero?

p;
                                     - x
(%o5)                             %e     cos(x)
```

We can also use the package **fourie.mac**, which we used above. This package provides for the calculation of our Fourier sine transform integral via the **fourintsin(expr,var)** function. The function **fourintsin(f,x)** returns an answer with the label b_z which contains the Fourier sine integral $F_S(f,\,z)$, with the letter **z** being the package convention for what we called **w** (ω).

```
(%i6) load(fourie);
(%o6) C:/PROGRA~1/MAXIMA~3.1/share/maxima/5.18.1/share/calculus/fourie.mac
(%i7) facts();
(%o7)                              [w > 0]
(%i8) (forget(w>0),facts());
(%o8)                                []
(%i9) fourintsin(f,x);
                                                3
                                           2 z
(%t9)                          b   = ------------
                                z             4
                                     %pi  (z   + 4)

(%o9)                              [%t9]
(%i10) bz : rhs(%t9);
                                           3
                                        2 z
(%o10)                            ------------
                                          4
                                  %pi  (z   + 4)
```

Thus **fourintsin(expr,var)** agrees with our definition of the Fourier sine transform.

10.3.3 Exponential Fourier Integrals and fourint

Given some function $\mathbf{f(x)}$ defined for $-\infty < \mathbf{x} < \infty$, we define the exponential Fourier transform of this function as

$$\mathbf{F_{Exp}(f,\omega) = \frac{1}{2\,\pi}\int_{-\infty}^{\infty} f(x) e^{i\,\omega\,x}\, dx} \tag{10.26}$$

The given function $\mathbf{f(x)}$ can then be written as an integral over both positive and negative values of $\boldsymbol{\omega}$:

$$\mathbf{f(x) = \int_{-\infty}^{\infty} F_{Exp}(f,\omega)\, e^{-i\,\omega\,x}\, d\omega} \tag{10.27}$$

The two equations (10.26) and (10.27) are another "Fourier transform pair", which include conventions about where to place the factor of $\mathbf{2\,\pi}$ as well as which member has the minus sign in the exponent.

If the given function is even, $\mathbf{f(-x) = f(x)}$, then the exponential Fourier transform has the symmetry

$$\mathbf{F_{Exp}(f,-\omega) = F_{Exp}(f,\omega)} \tag{10.28}$$

and can be expressed in terms of the Fourier cosine transform:

$$\mathbf{F_{Exp}(f,\omega) = \frac{1}{2}\, F_C(f,\omega)}. \tag{10.29}$$

If the given function is odd, $\mathbf{f(-x) = -f(x)}$, then the exponential Fourier transform has the symmetry

$$\mathbf{F_{Exp}(f,-\omega) = -F_{Exp}(f,\omega)} \tag{10.30}$$

and can be expressed in terms of the Fourier sine transform:

$$\mathbf{F_{Exp}(f,\omega) = \frac{i}{2}\, F_S(f,\omega)}. \tag{10.31}$$

If the given function is neither even nor odd, the function can always be written as the sum of an even function $\mathbf{f_e(x)}$ and an odd function $\mathbf{f_o(x)}$:

$$\mathbf{f(x) \equiv f_e(x) + f_o(x) = \frac{1}{2}\,(f(x) + f(-x) + \frac{1}{2}\,(f(x) - f(-x)} \tag{10.32}$$

and can be expressed in terms of the Fourier cosine transform of $\mathbf{f_e(x)}$ and the Fourier sine transform of $\mathbf{f_o(x)}$:

$$\mathbf{F_{Exp}(f,\omega) \equiv F_{Exp}(f_e + f_o,\omega) = \frac{1}{2}\, F_C(f_e,\omega) + \frac{i}{2}\, F_S(f_o,\omega)} \tag{10.33}$$

The **fourie.mac** package function **fourint(expr,var)** displays the non-zero coefficients $\mathbf{a_z}$ and/or the $\mathbf{b_z}$, in terms of which we can write down the value of $\mathbf{F_{Exp}(f,z)}$ (with our conventions):

$$\mathbf{F_{Exp}(f,z) = \frac{1}{2}\, a_z + \frac{i}{2}\, b_z} \tag{10.34}$$

10.3.4 Example 1: Even Function

We calculate here the exponential fourier transform $\mathbf{F_{Exp}(f,\omega)}$ when the function is $\mathbf{f(x) = \cos(x)\, e^{-|x|}}$ which is an even function $\mathbf{f(-x) = f(x)}$ and is defined for $-\infty < \mathbf{x} < \infty$.

We first use **integrate**, by separating the integral into two pieces, **i1** is the integral over $(-\infty, \mathbf{0})$ and **i2** is the integral over $(\mathbf{0}, \infty)$ (we ignore the overall factor of $\mathbf{1/(2\,\pi)}$ initially).

```
(%i1) assume(w>0)$
(%i2) i1:integrate(exp(%i*w*x)*cos(x)*exp(x),x,minf,0);
                                   2            3
                                  w  + 2    %i w
(%o2)                             ------ - ------
                                   4         4
                                  w  + 4    w  + 4
(%i3) i2:integrate(exp(%i*w*x)*cos(x)*exp(-x),x,0,inf);
                                      3      2
                                  %i w      w  + 2
(%o3)                             ------ + ------
                                   4         4
                                  w  + 4    w  + 4
(%i4) i12:ratsimp(i1+i2);
                                         2
                                     2 w  + 4
(%o4)                                --------
                                       4
                                      w  + 4
(%i5) 2*ratsimp(i12/2);
                                         2
                                    2 (w  + 2)
(%o5)                               ----------
                                       4
                                      w  + 4
(%i6) iexp:%/(2*%pi);
                                       2
                                      w  + 2
(%o6)                              ------------
                                         4
                                   %pi (w  + 4)
```

Hence, direct use of **integrate** provides the exponential Fourier transform

$$F_{Exp}(\cos(x)\exp(-|x|),\,\omega) = \frac{\omega^2 + 2}{\pi\,(\omega^4 + 4)} \tag{10.35}$$

We can use **integrate** to calculate the inverse Fourier exponential transform, recovering our original even function of x.

```
(%i7) integrate(exp(-%i*w*x)*iexp,w,minf,inf);
Is  x  positive, negative, or zero?
p;
                                     - x
(%o7)                              %e     cos(x)
(%i8) integrate(exp(-%i*w*x)*iexp,w,minf,inf);
Is  x  positive, negative, or zero?
n;
                                      x
(%o8)                               %e  cos(x)
```

Next we use **integrate** again directly to calculate the Fourier cosine transform of the given even function, now considered as defined for $x \geq 0$ and confirm that the required exponential Fourier transform in this case is correctly given by $\frac{1}{2} F_C(f, \omega)$.

```
(%i9) i3:ratsimp(integrate(cos(x)*exp(-x)*cos(w*x),x,0,inf));
                                     2
                                    w  + 2
(%o9)                               ------
                                     4
                                    w  + 4
(%i10) i3:(2/%pi)*i3;
                                      2
                                  2 (w  + 2)
(%o10)                           ------------
                                        4
                                 %pi (w  + 4)
```

Output `%o10` is the value of $F_C(f, \omega)$, and one half of that value is the required exponential Fourier transform.

Next we use **fourint(expr,var)** with this even function.

```
(%i1) load(fourie);
(%o1) C:/PROGRA~1/MAXIMA~3.1/share/maxima/5.18.1/share/calculus/fourie.mac
(%i2) fourint(cos(x)*exp(-abs(x)),x);
                                      2
                                     z        2
                                2 (------ + ------)
                                    4        4
                                   z  + 4   z  + 4
(%t2)                      a  = -------------------
                            z           %pi

(%t3)                               b  = 0
                                     z

(%o3)                             [%t2, %t3]
(%i4) ratsimp(rhs(%t2));
                                      2
                                   2 z  + 4
(%o4)                           --------------
                                     4
                                %pi z  + 4 %pi
(%i5) (2/%pi)*ratsimp(%pi*%/2);
                                      2
                                  2 (z  + 2)
(%o5)                            ------------
                                        4
                                 %pi (z  + 4)
```

which confirms that for an even function, the required exponential Fourier transform is correctly given (using **fourint**) by $F_{Exp}(f, z) = a_z/2$.

10.3.5 Example 2: Odd Function

We calculate here the exponential fourier transform $\mathbf{F_{Exp}}(\mathbf{f}, \omega)$ when the function is $\mathbf{f}(\mathbf{x}) = \sin(\mathbf{x})\,\mathbf{e}^{-|\mathbf{x}|}$ which is an odd function $\mathbf{f}(-\mathbf{x}) = -\mathbf{f}(\mathbf{x})$ and is defined for $-\infty < \mathbf{x} < \infty$.

We first use **integrate**, by separating the integral into two pieces, **i1** is the integral over $(-\infty, \mathbf{0})$ and **i2** is the integral over $(\mathbf{0}, \infty)$ (we ignore the overall factor of $\mathbf{1}/(\mathbf{2}\,\pi)$ initially).

```
(%i1) ( assume(w>0), facts());
(%o1)                               [w > 0]
(%i2) i1:ratsimp(integrate(exp(%i*w*x)*sin(x)*exp(x),x,minf,0));
                                2
                               w  + 2 %i w - 2
(%o2)                          ---------------
                                   4
                                  w  + 4
(%i3) i2:ratsimp(integrate(exp(%i*w*x)*sin(x)*exp(-x),x,0,inf));
                                 2
                                w  - 2 %i w - 2
(%o3)                        - ---------------
                                    4
                                   w  + 4
(%i4) iexp:ratsimp(i1+i2)/(2*%pi);
                                  2 %i w
(%o4)                           ------------
                                     4
                                %pi (w  + 4)
(%i5) facts();
(%o5)                              [w > 0]
```

Hence, direct use of **integrate** provides the exponential Fourier transform

$$\mathbf{F_{Exp}}(\sin(\mathbf{x})\exp(-|\mathbf{x}|),\,\omega) = \frac{2\,\mathrm{i}\,\omega}{\pi\,(\omega^4 + 4)} \tag{10.36}$$

We can use **integrate** to calculate the inverse Fourier exponential transform, recovering our original odd function of $\mathbf{x}$.

```
(%i6) integrate(exp(-%i*w*x)*iexp,w,minf,inf);
Is  x  positive, negative, or zero?
p;
                                  - x
(%o6)                           %e    sin(x)
(%i7) integrate(exp(-%i*w*x)*iexp,w,minf,inf);
Is  x  positive, negative, or zero?
n;
                                   x
(%o7)                            %e  sin(x)
(%i8) facts();
(%o8)                              [w > 0]
```

Next we use **integrate** again directly to calculate the Fourier sine transform of the given odd function, now considered as defined for $x \geq 0$ and confirm that the required exponential Fourier transform in this case is correctly given by $\frac{i}{2}\,F_S(f, \omega)$.

```
(%i9) ratsimp(integrate(sin(x)*exp(-x)*sin(w*x),x,0,inf));
                                  2 w
(%o9)                            ------
                                  4
                                 w  + 4
(%i10) (2/%pi)*%;
                                  4 w
(%o10)                        ------------
                                    4
                              %pi (w  + 4)
```

Output `%o10` is the value of $F_S(f, \omega)$, and multiplying by $i/2$ yields the required exponential Fourier transform.

Next we use **fourint(expr,var)** with this odd function.

```
(%i11) load(fourie);
(%o11) C:/PROGRA~1/MAXIMA~3.1/share/maxima/5.18.1/share/calculus/fourie.mac
(%i12) fourint(sin(x)*exp(-abs(x)),x);
(%t12)                           a  = 0
                                  z

                                     4 z
(%t13)                      b  = ------------
                             z          4
                                 %pi (z  + 4)

(%o13)                        [%t12, %t13]
```

which confirms that for an odd function, the required exponential Fourier transform is correctly given (using **fourint**) by $F_{Exp}(f, z) = (i\, b_z)/2$.

10.3.6 Example 3: A Function Which is Neither Even nor Odd

We calculate here the exponential fourier transform $\mathbf{F_{Exp}}(\mathbf{f}, \boldsymbol{\omega})$ when the function is $\mathbf{f}(\mathbf{x}) = \cos^{\mathbf{2}}(\mathbf{x} - \mathbf{1})\, \mathbf{e}^{-|\mathbf{x}|}$ which is defined for $-\infty < \mathbf{x} < \infty$ and is neither even nor odd.

We first use **integrate**, by separating the integral into two pieces. **i1** is the integral over $(-\infty, \mathbf{0})$ and **i2** is the integral over $(\mathbf{0}, \infty)$ (we ignore the overall factor of $\mathbf{1}/(\mathbf{2}\,\boldsymbol{\pi})$ initially).

```
(%i1) ( assume(w>0), facts());
(%o1)                               [w > 0]
(%i2) i1:ratsimp(integrate(exp(%i*w*x)*cos(x-1)^2*exp(x),x,minf,0));
                                   5                               4
(%o2) - ((%i cos(2) + %i) w  + (- 2 sin(2) - cos(2) - 1) w
                                              3                              2
 + (- 4 %i sin(2) - 2 %i cos(2) - 6 %i) w  + (8 sin(2) - 6 cos(2) + 6) w
 + (- 4 %i sin(2) - 3 %i cos(2) + 25 %i) w + 10 sin(2) - 5 cos(2) - 25)
     6        4        2
/(2 w  - 10 w  + 38 w  + 50)
(%i3) i2:ratsimp(integrate(exp(%i*w*x)*cos(x-1)^2*exp(-x),x,0,inf));
                                 5                               4
(%o3) ((%i cos(2) + %i) w  + (- 2 sin(2) + cos(2) + 1) w
                                            3                              2
 + (4 %i sin(2) - 2 %i cos(2) - 6 %i) w  + (8 sin(2) + 6 cos(2) - 6) w
 + (4 %i sin(2) - 3 %i cos(2) + 25 %i) w + 10 sin(2) + 5 cos(2) + 25)
     6        4        2
/(2 w  - 10 w  + 38 w  + 50)
(%i4) i12 : rectform(ratsimp(i1 + i2));
                         4                    2
      (cos(2) + 1) w  + (6 cos(2) - 6) w  + 5 cos(2) + 25
(%o4) ---------------------------------------------------
                          6       4        2
                         w  - 5 w  + 19 w  + 25
                                                                     3
                                                    %i (4 sin(2) w  + 4 sin(2) w)
                                                  + -----------------------------
                                                          6       4        2
                                                         w  - 5 w  + 19 w  + 25
(%i5) i12 : map('ratsimp,i12);
                         4                    2
      (cos(2) + 1) w  + (6 cos(2) - 6) w  + 5 cos(2) + 25    4 %i sin(2) w
(%o5) --------------------------------------------------- + --------------
                          6       4        2                    4      2
                         w  - 5 w  + 19 w  + 25               w  - 6 w  + 25
(%i6) i12 : realpart(i12)/(2*%pi) + %i*imagpart(i12)/(2*%pi);
                         4                    2
      (cos(2) + 1) w  + (6 cos(2) - 6) w  + 5 cos(2) + 25
(%o6) ---------------------------------------------------
                             6       4        2
                    2 %pi (w  - 5 w  + 19 w  + 25)
                                                             2 %i sin(2) w
                                                           + --------------------
                                                                    4      2
                                                              %pi (w  - 6 w  + 25)
```

The final `i12` is the desired exponential Fourier transform.

We can now calculate the inverse Fourier transform.

```
(%i7) integrate(exp(-%i*w*x)*i12,w,minf,inf);
Is  x  positive, negative, or zero?
p;
                      - x
                    %e    (sin(2) sin(2 x) + cos(2) cos(2 x) + 1)
(%o7)               ---------------------------------------------
                                          2
```

As an exercise in the manipulation of trigonometric functions, we now go back and forth between the coefficient of `%e^(-x)` in this result and our starting function (for positive x).

```
(%i8) cos(x-1)^2;
                                    2
(%o8)                            cos  (x - 1)
(%i9) trigreduce(%);
                             cos(2 (x - 1)) + 1
(%o9)                        ------------------
                                     2

(%i10) ratsimp(%);
                              cos(2 x - 2) + 1
(%o10)                        ----------------
                                     2

(%i11) trigexpand(%);
                     sin(2) sin(2 x) + cos(2) cos(2 x) + 1
(%o11)               -------------------------------------
                                       2
```

which gets us from the original coefficient to that returned by our inverse transform integral. Now we go in the opposite direction:

```
(%i12) trigreduce(%);
                              cos(2 x - 2) + 1
(%o12)                        ----------------
                                     2

(%i13) factor(%);
                             cos(2 (x - 1)) + 1
(%o13)                       ------------------
                                     2

(%i14) trigexpand(%);
                             2              2
                       - sin  (x - 1) + cos  (x - 1) + 1
(%o14)                 -------------------------------
                                       2

(%i15) trigsimp(%);
                                    2
(%o15)                           cos  (x - 1)
```

which returns us from the transform integral coefficient to our original coefficient. Hence we have verified the correctness of the exponential Fourier transform for this case.

Next we use **fourint(expr,var)** with this function which is neither even nor odd.

```
(%i16) load(fourie);
(%o16) C:/PROGRA~1/MAXIMA~3.1/share/maxima/5.18.1/share/calculus/fourie.mac
(%i17) fourint(cos(x-1)^2*exp(-abs(x)),x);
                            4                      2
             (cos(2) + 1) z  + (6 cos(2) - 6) z  + 5 cos(2) + 25
(%t17)     a  = ---------------------------------------------------
            z                 6      4       2
                       %pi (z  - 5 z  + 19 z  + 25)

                                    4 sin(2) z
(%t18)                     b  = --------------------
                            z          4      2
                                  %pi (z  - 6 z  + 25)

(%o18)                           [%t17, %t18]
(%i19) az : rhs(%t17);
                          4                      2
           (cos(2) + 1) z  + (6 cos(2) - 6) z  + 5 cos(2) + 25
(%o19)     ---------------------------------------------------
                              6      4       2
                       %pi (z  - 5 z  + 19 z  + 25)
(%i20) bz : rhs(%t18);
                                 4 sin(2) z
(%o20)                      --------------------
                                  4      2
                            %pi (z  - 6 z  + 25)
(%i21) iexp_f : az/2 + %i*bz/2;
                    4                      2
       (cos(2) + 1) z  + (6 cos(2) - 6) z  + 5 cos(2) + 25
(%o21) ---------------------------------------------------
                          6      4       2
                2 %pi (z  - 5 z  + 19 z  + 25)
                                                          2 %i sin(2) z
                                                      + --------------------
                                                              4      2
                                                        %pi (z  - 6 z  + 25)
```

To compare this with our result using **integrate** we need to replace **z** by **w**:

```
(%i22) subst(z=w,iexp_f) - i12;
(%o22)                             0
```

which shows that, except for notation, the results are the same. We have confirmed that for a general function, the required exponential Fourier transform is correctly given (using **fourint**) by $\mathbf{F_{Exp}(f,z) = a_z/2 + (i\,b_z)/2}$.

10.3.7 Dirac Delta Function $\delta(x)$

It is conventional to define the Dirac delta function (unit impulse function) $\delta(x)$ by

$$\delta(-x) = \delta(x) \quad \text{even function} \tag{10.37}$$

$$\delta(x) = 0 \quad \text{for} \quad x \neq 0 \tag{10.38}$$

$$\int_{-a}^{b} \delta(x)\, dx = 1 \quad \text{for} \quad a,\, b > 0 \tag{10.39}$$

These equations imply

$$\delta(y-x) = \delta(x-y) \tag{10.40}$$

and

$$\int f(y)\, \delta(y-x)\, dy = f(x) \tag{10.41}$$

for any well behaved function $f(x)$ (we are more careful below), provided the range of integration includes the point x. For since $\delta(y-x)$ is zero except at the single point $y = x$, we can evaluate $f(y)$ at that single point and take the result outside the integral. The resulting integral which is left is $\int \delta(y-x)\, dy$, and the change of variables $y \to t = y - x$ for fixed x yields the integral $\int \delta(t)\, dt = 1$.

Now let c be a positive constant (independent of the integration variable x). In the integral $\int_{-a}^{b} \delta(c\, x)\, dx$, we change variables, $x \to y = c\, x$ so $dx = dy/c$.

$$\int_{-a}^{b} \delta(c\, x)\, dx = \frac{1}{c} \int_{-c\, a}^{c\, b} \delta(y)\, dy = \frac{1}{c} \tag{10.42}$$

since $(c\, a) > 0$ and $(c\, b) > 0$.

If, on the other hand, c is a negative constant, $c = -|c|$, and we make the same change of variables,

$$\int_{-a}^{b} \delta(c\, x)\, dx = \frac{1}{c} \int_{-c\, a}^{c\, b} \delta(y)\, dy = \frac{1}{c} \int_{|c|\, a}^{-|c|\, b} \delta(y)\, dy = -\frac{1}{c} \int_{-|c|\, b}^{|c|\, a} \delta(y)\, dy = \frac{1}{|c|} \tag{10.43}$$

Evidently, we can always write (since both x and y are dummy integration variables

$$\int \delta(c\, x)\, dx = \int \frac{\delta(x)}{|c|}\, dx \tag{10.44}$$

or simply (with the understanding that this kind of relation only makes sense inside an integral)

$$\delta(c\, x) = \frac{\delta(x)}{|c|} \tag{10.45}$$

We can find a sometimes useful representation of the Dirac delta function by using the exponential Fourier transform pair Eqs.(10.26) and (10.27). If we use y as the (dummy) variable of integration in Eq.(10.26), and insert into Eq.(10.27), we get (interchanging the order of integration)

$$\begin{aligned} f(x) &= \int_{-\infty}^{\infty} \left[\frac{1}{2\,\pi} \int_{-\infty}^{\infty} e^{i\,\omega\, y}\, f(y)\, dy \right] e^{-i\,\omega\, x}\, d\omega \\ &= \int_{-\infty}^{\infty} f(y) \left[\frac{1}{2\,\pi} \int_{-\infty}^{\infty} e^{i\,\omega(y-x)}\, d\omega \right] dy \\ &= \int_{-\infty}^{\infty} f(y)\, \delta(y-x)\, dy \end{aligned}$$

Hence we can write

$$\delta(x) = \frac{1}{2\,\pi} \int_{-\infty}^{\infty} e^{i\,\omega\, x}\, d\omega \tag{10.46}$$

One use of such a representation of the one dimensional Dirac delta function is to derive in a different way the result of Eq.(10.45). From Eq.(10.46) we have

$$\delta(c\, x) = \frac{1}{2\,\pi} \int_{-\infty}^{\infty} e^{i\,\omega\, c\, x}\, d\omega \tag{10.47}$$

Again, suppose $\mathbf{c}$ is a constant and $\mathbf{c} > \mathbf{0}$. Then if we change variables $\boldsymbol{\omega} \to \mathbf{y} = \boldsymbol{\omega}\,\mathbf{c}$, we have $\mathbf{d}\boldsymbol{\omega} = \mathbf{dy}/\mathbf{c}$. When $\boldsymbol{\omega} = -\infty, \mathbf{y} = -\infty$. When $\boldsymbol{\omega} = \infty, \mathbf{y} = \infty$. Hence

$$\boldsymbol{\delta}(\mathbf{c}\,\mathbf{x}) = \frac{\mathbf{1}}{\mathbf{c}} \left(\frac{\mathbf{1}}{\mathbf{2}\,\boldsymbol{\pi}} \int_{-\infty}^{\infty} \mathbf{e}^{\mathbf{i}\,\mathbf{y}\,\mathbf{x}}\, \mathbf{dy} \right) = \frac{\mathbf{1}}{\mathbf{c}}\, \boldsymbol{\delta}(\mathbf{x}) \tag{10.48}$$

If, on the other hand, $\mathbf{c} < \mathbf{0}$, $\mathbf{c} = -|\mathbf{c}|$ in Eq.(10.47), and we make the same change of variables, when $\boldsymbol{\omega} = -\infty$, $\mathbf{y} = \infty$, and when $\boldsymbol{\omega} = \infty$, $\mathbf{y} = -\infty$. Hence, for this case,

$$\boldsymbol{\delta}(\mathbf{c}\,\mathbf{x}) = \frac{\mathbf{1}}{\mathbf{c}} \left(\frac{\mathbf{1}}{\mathbf{2}\,\boldsymbol{\pi}} \int_{\infty}^{-\infty} \mathbf{e}^{\mathbf{i}\,\mathbf{y}\,\mathbf{x}}\, \mathbf{dy} \right) = \frac{\mathbf{1}}{\mathbf{c}} \left(-\frac{\mathbf{1}}{\mathbf{2}\,\boldsymbol{\pi}} \int_{-\infty}^{\infty} \mathbf{e}^{\mathbf{i}\,\mathbf{y}\,\mathbf{x}}\, \mathbf{dy} \right) = \frac{\mathbf{1}}{|\mathbf{c}|}\, \boldsymbol{\delta}(\mathbf{x}). \tag{10.49}$$

Combining the two cases leads us again to Eq.(10.45).

Although the representation Eq.(10.46) can be useful for formal properties of the Dirac delta function, for actual calculation of integrals one is safer using a limiting form of that result, or even simpler versions, as we will see. If we replace the infinite limits in Eq.(10.46) by finite limits, we can work with the representation

$$\boldsymbol{\delta}(\mathbf{x}) = \lim_{L \to \infty} \mathbf{d_L}(\mathbf{x}) \tag{10.50}$$

where

$$\mathbf{d_L}(\mathbf{x}) = \frac{\mathbf{1}}{\mathbf{2}\,\boldsymbol{\pi}} \int_{-\mathbf{L}}^{\mathbf{L}} \mathbf{e}^{\mathbf{i}\,\mathbf{x}\,\mathbf{y}}\, \mathbf{dy} = \frac{\sin(\mathbf{x}\,\mathbf{L})}{\boldsymbol{\pi}\,\mathbf{x}} \tag{10.51}$$

which can then be used as

$$\int \mathbf{f}(\mathbf{x})\, \boldsymbol{\delta}(\mathbf{x})\, \mathbf{dx} = \lim_{L \to \infty} \int \mathbf{f}(\mathbf{x})\, \mathbf{d_L}(\mathbf{x})\, \mathbf{dx} = \lim_{L \to \infty} \int \mathbf{f}(\mathbf{x})\, \frac{\sin(\mathbf{x}\,\mathbf{L})}{\boldsymbol{\pi}\,\mathbf{x}}. \tag{10.52}$$

However, this will only be useful if the resulting integrals can be done.

An easier approach to the careful use of the Dirac delta function for the calculation of integrals is to create a mathematically simple model of a function of $\mathbf{x}$ which has the required properties and whose use will result in integrals which can easily be done. Mathematically rigorous treatments (see the Theory of Distributions) of the Dirac delta function justify the use of such models .

Here we follow the approach of William E. Boyce and Richard C. DiPrima, in their textbook "Elementary Differential Equations and Boundary Value Problems", Fifth Edition, John Wiley & Sons, Inc, New York, 1992, Section 6.5, "Impulse Functions".

Let

$$\mathbf{d}_{\boldsymbol{\epsilon}}(\mathbf{x}) = \left\{ \begin{array}{cll} 1/(2\,\boldsymbol{\epsilon}) & \text{for} & -\boldsymbol{\epsilon} < \mathbf{x} < \boldsymbol{\epsilon} \\ 0 & \text{for} & \mathbf{x} \le -\boldsymbol{\epsilon} \quad \text{or} \quad \mathbf{x} \ge \boldsymbol{\epsilon} \end{array} \right\} \tag{10.53}$$

which defines a rectangular pulse with base length $\mathbf{2}\,\boldsymbol{\epsilon}$, height $\mathbf{1}/(\mathbf{2}\,\boldsymbol{\epsilon})$ with area "under the curve" equal to $\mathbf{1}$. This is an even function of $\mathbf{x}$ centered on $\mathbf{x} = \mathbf{0}$ which gets narrower and taller as $\boldsymbol{\epsilon} \to \mathbf{0}$.

Then

$$\mathbf{d}_{\boldsymbol{\epsilon}}(\mathbf{y} - \mathbf{x}) = \left\{ \begin{array}{cll} 1/(2\,\boldsymbol{\epsilon}) & \text{for} & (\mathbf{x} - \boldsymbol{\epsilon}) < \mathbf{y} < (\mathbf{x} + \boldsymbol{\epsilon}) \\ 0 & \text{for} & \mathbf{y} \le (\mathbf{x} - \boldsymbol{\epsilon}) \quad \text{or} \quad \mathbf{y} \ge (\mathbf{x} + \boldsymbol{\epsilon}) \end{array} \right\} \tag{10.54}$$

This model can then be used in the form (with the range of integration assumed to include the point $y = x$):

$$\int f(y)\,\delta(y-x)\,dy = \lim_{\epsilon \longrightarrow 0} \int f(y)\,d_\epsilon(y-x)\,dy = \lim_{\epsilon \longrightarrow 0} \frac{1}{2\,\epsilon} \int_{x-\epsilon}^{x+\epsilon} f(y)\,dy. \tag{10.55}$$

The integral should be done before taking the limits in this safer approach.

However, if $f(y)$ is a continuous function and possesses a derivative in a complete neighborhood of x, we can use the mean value theorem of calculus to write this limit as

$$\lim_{\epsilon \longrightarrow 0} \frac{1}{2\,\epsilon} f(\hat{y})\,(2\,\epsilon) = \lim_{\epsilon \longrightarrow 0} f(\hat{y})). \tag{10.56}$$

where $(x-\epsilon) < \hat{y} < (x+\epsilon)$ As $\epsilon \longrightarrow 0$, $\hat{y} \to x$, and $f(\hat{y}) \to f(x)$, which recovers Eq. (10.41), and which we repeat here for emphasis.

$$\int f(y)\,\delta(y-x)\,dy = f(x) \tag{10.57}$$

for any function $f(x)$ which is continuous and possesses a derivative in a complete neighborhood of x, provided the range of integration includes the point x.

10.3.8 Laplace Transform of the Delta Function Using a Limit Method

Use of Eq. (10.57) allows us to immediately write down the Laplace transform of $\delta(t-t_0)$ for $t_0 > 0$, the integral $\int_0^\infty e^{-s\,t}\,\delta(t-t_0)\,dt$, since the function $e^{-s\,t}$ which multiplies the delta function is continuous and possesses a derivative for all values of t.

However, as a consistency check, let's also use the limit method just discussed to derive this result.

$$\mathcal{L}\{\delta(t-t_0)\} \equiv \int_0^\infty e^{-s\,t}\,\delta(t-t_0)\,dt = e^{-s\,t_0} \tag{10.58}$$

provided s and t_0 are positive.

```
(%i1) assume(s>0,e>0,t0>0)$
(%i2) i1: integrate(exp(-s*t),t,t0-e,t0+e)/(2*e);
                                e s - s t0      - s t0 - e s
                              %e              %e
                              ------------ - --------------
                                   s               s
(%o2)                         -----------------------------
                                           2 e
(%i3) limit(i1,e,0,plus);
                                         - s t0
(%o3)                                  %e
```

As a bonus, the limit of Eq.(10.58) as $t_0 \to 0^+$ then gives us the Laplace transform of $\delta(t)$.

$$\mathcal{L}\{\delta(t)\} \equiv \int_0^\infty e^{-s\,t}\,\delta(t)\,dt = 1 \tag{10.59}$$

10.4 Laplace Transform Integrals

10.4.1 Laplace Transform Integrals: laplace(..), specint(..)

A subgroup of definite integrals is so useful in applications that Maxima has functions which calculate the Laplace transform (**laplace** and **specint**) of a given expression, as well as the inverse Laplace transform (**ilt**).

laplace(expr, t, s) calculates the integral

$$\int_0^\infty e^{-s\,t}\, f(t)\, d\,t = \mathcal{L}\{f(t)\} = F(s) \tag{10.60}$$

where $f(t)$ stands for the **expr** argument (here assumed to depend explicitly on t) of **laplace**. **laplace** assumes that s is a very large positive number. If there are convergence issues of the integral, one allows s to be a complex number $s = x + i\,y$ and the assumption is made that the integral converges to the right of some line $\mathrm{Re}(s) = \mathbf{const}$ in the complex s plane.

10.4.2 Comparison of laplace and specint

specint(exp(- s*t) * expr, t) is equivalent to **laplace(expr, t, s)** in its effect, but has been specially designed to handle special functions with care. You need to prepare **specint** with **assume** statement(s) that s is both positive and larger than other positive parameters in **expr**.

$\mathcal{L}\{1\} = 1/s$

We start the following comparison of **laplace** and **specint** with an **assume** statement (needed by **specint** alone).

```
(%i1) assume( s>0, a > 0, b > 0, s > a, s > b )$
(%i2) laplace(1,t,s);
                                         1
(%o2)                                    -
                                         s
(%i3) specint(exp(-s*t),t);
                                         1
(%o3)                                    -
                                         s
(%i4) ilt(%,s,t);
(%o4)                                    1
```

$\mathcal{L}\{t\} = 1/s^2$

```
(%i5) laplace(t,t,s);
                                        1
(%o5)                                  --
                                         2
                                        s
(%i6) specint(exp(-s*t)*t,t);
                                        1
(%o6)                                  --
                                         2
                                        s
(%i7) ilt(%,s,t);
(%o7)                                    t
```

$\mathcal{L}\{e^{a\,t}\} = 1/(s-a)$

```
(%i8) laplace(exp(a*t),t,s);
                                              1
(%o8)                                       -----
                                            s - a
(%i9) specint(exp(-s*t)*exp(a*t),t);
                                              1
(%o9)                                       -----
                                            s - a
```

$\mathcal{L}\{\sin(a\,t)/a\} = (s^2+a^2)^{-1}$

```
(%i10) laplace(sin(a*t)/a,t,s);
                                              1
(%o10)                                     -------
                                            2    2
                                           s  + a
(%i11) (specint(exp(-s*t)*sin(a*t)/a,t),ratsimp(%%) );
                                              1
(%o11)                                     -------
                                            2    2
                                           s  + a
```

$\mathcal{L}\{\cos(a\,t)\} = s\,(s^2+a^2)^{-1}$

```
(%i12) laplace(cos(a*t),t,s);
                                              s
(%o12)                                     -------
                                            2    2
                                           s  + a
(%i13) (specint(exp(-s*t)*cos(a*t),t ), ratsimp(%%) );
                                              s
(%o13)                                     -------
                                            2    2
                                           s  + a
```

$\mathcal{L}\{t\sin(a\,t)/(2\,a)\} = s\,(s^2+a^2)^{-2}$

```
(%i14) laplace(sin(a*t)*t/(2*a),t,s);
                                             s
(%o14)                                   ----------
                                           2    2 2
                                         (s  + a )
(%i15) (specint(exp(-s*t)*sin(a*t)*t/(2*a),t ),ratsimp(%%) );
                                              s
(%o15)                                -----------------
                                       4      2  2    4
                                      s  + 2 a  s  + a
(%i16) map('factorsum,%);
                                             s
(%o16)                                   ----------
                                           2    2 2
                                         (s  + a )
```

$\mathcal{L}\{e^{a\,t}\cos(b\,t)\} = (s-a)\,(\,(s-a)^2 + b^2\,)^{-1}$

```
(%i17) laplace(exp(a*t)*cos(b*t),t,s);
                                     s - a
(%o17)                       --------------------
                              2            2    2
                             s  - 2 a s + b  + a
(%i18) map('factorsum,%);
                                      s - a
(%o18)                           -------------
                                        2    2
                                 (s - a)  + b
(%i19) (specint(exp(-s*t)*exp(a*t)*cos(b*t),t ),ratsimp(%%) );
                                     s - a
(%o19)                       --------------------
                              2            2    2
                             s  - 2 a s + b  + a
(%i20) map('factorsum,%);
                                      s - a
(%o20)                           -------------
                                        2    2
                                 (s - a)  + b
```

$\mathcal{L}\{\sqrt{t}\ \textbf{bessel_j}\ (1, 2\,\sqrt{a\,t})\} = \sqrt{a}\,s^{-2}\,e^{-a/s}$

```
(%i21) expr : t^(1/2) * bessel_j(1, 2 * a^(1/2) * t^(1/2));
(%o21)               bessel_j(1, 2 sqrt(a) sqrt(t)) sqrt(t)
(%i22) laplace(expr,t,s);
                                          - a/s
                                 sqrt(a) %e
(%o22)                           ---------------
                                         2
                                        s
(%i23) specint(exp(-s*t)*expr,t);
                                          - a/s
                                 sqrt(a) %e
(%o23)                           ---------------
                                         2
                                        s
(%i24) ilt(%,s,t);
                                        - a/s
                               sqrt(a) %e
(%o24)                     ilt(---------------, s, t)
                                       2
                                      s
```

We find in the above example that **ilt** cannot find the inverse Laplace transform.

$\mathcal{L}\{\mathrm{erf}(\sqrt{t})\} = (\,s\,\sqrt{s+1}\,)^{-1}$

Here again **ilt** fails.

```
(%i25) assume(s>0)$
(%i26) laplace(erf(sqrt(t)),t,s);
                                          1
(%o26)                              -------------
                                    s sqrt(s + 1)
(%i27) specint(exp(-s*t)*erf(sqrt(t)),t);
                                         1
(%o27)                           ----------------
                                      1       3/2
                                 sqrt(- + 1) s
                                      s
(%i28) radcan(%);
                                          1
(%o28)                              -------------
                                    s sqrt(s + 1)
(%i29) ilt(%,s,t);
                                        1
(%o29)                       ilt(-------------, s, t)
                                 s sqrt(s + 1)
```

$\mathcal{L}\{\mathrm{erf}(t)\} = e^{s^2/4}\,(1 - \mathrm{erf}(s/2))\,s^{-1}$

Here **laplace** succeeds, and **ilt** and **specint** fail.

```
(%i30) laplace(erf(t),t,s);
                                     2
                                    s
                                    --
                                    4            s
                                 %e    (1 - erf(-))
                                                 2
(%o30)                           -----------------
                                         s
(%i31) ilt(%,s,t);
                                   2
                                  s
                                  --
                                  4            s
                               %e    (1 - erf(-))
                                               2
(%o31)                     ilt(-----------------, s, t)
                                       s
(%i32) specint(exp(-s*t)*erf(t),t);
                                      - s t
(%o32)                      specint(%e      erf(t), t)
```

10.4.3 Use of the Dirac Delta Function (Unit Impulse Function) delta with laplace(..)

The **laplace** function recognises **delta(arg)** as part of the expression supplied to **laplace**. We have introduced the Dirac delta function (unit impulse function) in Sections 10.3.7 and 10.3.8. Here are three examples of using **delta** with **laplace**.

$\mathcal{L}\{\delta(t)\} = 1$

```
(%i1) laplace(delta(t),t,s);
(%o1)                                     1
(%i2) ilt(1,s,t);
(%o2)                              ilt(1, s, t)
```

$\mathcal{L}\{\delta(t-a)\} = e^{-a\,s}$

(for $a > 0$):

```
(%i3) laplace(delta(t-a),t,s);
Is  a  positive, negative, or zero?
p;
                                        - a s
(%o3)                                 %e
```

$\mathcal{L}\{\delta(t-a)\sin(b\,t)\} = \sin(a\,b)\,e^{-a\,s}$

(for $a > 0$):

```
(%i4) laplace(delta(t-a)*sin(b*t),t,s);
Is  a  positive, negative, or zero?
p;
                                            - a s
(%o4)                            sin(a b) %e
```

In this last example, we see that **laplace** is simply using

$$\int_0^{\infty} f(t)\,\delta(t-a)\,dt = f(a) \qquad (10.61)$$

with $a > 0$ and $f(t) = e^{-s\,t}\sin(b\,t)$.

10.5 The Inverse Laplace Transform and Residues at Poles

10.5.1 ilt: Inverse Laplace Transform

The Maxima function: **ilt (expr, s, t)** computes the inverse Laplace transform of **expr** with respect to s and in terms of the parameter t. This Maxima function is able to compute the inverse Laplace transform only if **expr** is a rational algebraic function (a quotient of two polynomials).

In elementary work with Laplace transforms, one uses a "look-up table" of Laplace transforms, together with general properties of Laplace and inverse Laplace transforms, to determine inverse Laplace transforms of elementary and special functions.

For advanced work, one can use the general formula for the inverse Laplace transform

$$\mathbf{f}(\mathbf{t} > \mathbf{0}) = \frac{\mathbf{1}}{\mathbf{2}\,\pi\,\mathbf{i}} \int_{\sigma-\mathbf{i}\,\infty}^{\sigma+\mathbf{i}\,\infty} \mathbf{e}^{\mathbf{s}\,\mathbf{t}}\,\mathbf{F}(\mathbf{s})\,\mathbf{ds} \tag{10.62}$$

where the contour of integration is a vertical line in the complex $\mathbf{s} = \mathbf{x} + \mathbf{i}\,\mathbf{y}$ plane (ie., along a line $\mathbf{x} = \sigma$), to the right of all singularities (poles, branch points, and essential singularities) of the integrand. Here we assume the only type of singularities we need to worry about are poles, isolated values of $\mathbf{s}$ where the approximate power series expansion for $\mathbf{s}$ near $\mathbf{s_j}$ is $\mathbf{g}(\mathbf{s_j})/(\mathbf{s} - \mathbf{s_j})^{\mathbf{m}}$, where $\mathbf{g}(\mathbf{s_j})$ is a finite number, and $\mathbf{m}$ is the "order" of the pole. If $\mathbf{m} = \mathbf{1}$, we call it a "simple pole". At a particular point $(\mathbf{x}, \mathbf{y})$ of the contour, the exponential factor has the form $\mathbf{e}^{\mathbf{s}\,\mathbf{t}} = \mathbf{e}^{\mathbf{i}\,\mathbf{t}\,\mathbf{y}}\,\mathbf{e}^{\mathbf{t}\,\mathbf{x}}$. Because $\mathbf{t} > \mathbf{0}$ the integrand is heavily damped by the factor $\mathbf{e}^{\mathbf{x}\,\mathbf{t}}$ when $\mathbf{x} \to -\infty$, and the contour of integration can be turned into a closed contour by adding a zero contribution which is the integral of the same integrand along a large arc in the left hand $\mathbf{s}$ plane (in the limit that the radius of the arc approaches ∞), and the resulting closed contour integral can then be evaluated using the residue theorem of complex variable theory. The (counter-clockwise) closed contour integral is then given by $\mathbf{2}\,\pi\,\mathbf{i}$ times the sum of the residues of the integrand at the isolated poles enclosed by the contour.

The factor $\mathbf{2}\,\pi\,\mathbf{i}$ coming from the integral part of Eq.(10.62) is then cancelled by the factor $\mathbf{1}/(\mathbf{2}\,\pi\,\mathbf{i})$ which appears in the definition (Eq.(10.62)) of the inverse Laplace transform, and one has the simple result that **the inverse Laplace transform is the sum of the residues of the poles of the integrand**.

10.5.2 residue: Residues at Poles

We can use the Maxima **residue** function to illustrate how the sum of the residues at all the poles of the integrand reproduces the answer returned by **ilt**.

> **`residue (expr, z, z_0)`**
> Computes the residue in the complex plane of the expression **`expr`** when the variable $\mathbf{z}$ assumes the value **`z_0`**. The residue is the coefficient of **`(z - z_0)^(-1)`** in the Laurent series for **`expr`**.

```
(%i1) residue (s/(s^2+a^2), s, a*%i);
                                        1
(%o1)                                   -
                                        2
(%i2) residue (sin(a*x)/x^4, x, 0);
                                         3
                                        a
(%o2)                                 - --
                                        6
```

We start with some arbitrary rational algebraic function of $\mathbf{s}$, use **ilt** to determine the corresponding function of $\mathbf{t}$, and then compute the inverse Laplace transform (a second way) by summing the values of the residues of the Laplace inversion integrand in the complex $\mathbf{s}$ plane.

```
(%i3) fs : -s/(s^3 +4*s^2 + 5*s +2)$
(%i4) ft : ( ilt(fs,s,t),collectterms(%%,exp(-t),exp(-2*t) ) );
                                      - t        - 2 t
(%o4)                        (t - 2) %e    + 2 %e
```

```
(%i5) (laplace(ft,t,s), ratsimp(%%) );
                                          s
(%o5)                       - -------------------
                                3      2
                               s  + 4 s  + 5 s + 2
(%i6) fs - %;
(%o6)                                  0
(%i7) fst : exp(s*t)*fs;
                                           s t
                                       s %e
(%o7)                       - -------------------
                                3      2
                               s  + 4 s  + 5 s + 2
(%i8) partfrac(fst,s);
                             s t       s t      s t
                         2 %e      2 %e       %e
(%o8)                    ------- - ------- + --------
                          s + 2     s + 1           2
                                                (s + 1)
```

The inverse Laplace transform **integrand** above was called **fst**, and we used **partfrac** on **fst** to exhibit the poles of this integrand. We see that there is one simple pole at $s = -2$ and one pole of order 2 located at $s = -1$.

To use the Maxima function **residue**, we need to tell Maxima that $t > 0$.

```
(%i9) assume( t > 0 )$
(%i10) r1 : residue(fst,s,-2);
                                          - 2 t
(%o10)                               2 %e
(%i11) r2 : residue(fst,s,-1);
                                              - t
(%o11)                            (t - 2) %e
(%i12) r1 + r2 - ft;
(%o12)                                 0
```

which shows that the sum of the residues reproduces the function of t which **ilt** produced.

This concludes our discussion of the inverse Laplace transform.

Maxima by Example:
Ch.11: Fast Fourier Transform Tools *

Edwin L. Woollett

August 13, 2009

Contents

11.1 Examples of the Use of the Fast Fourier Transform Functions **fft** and **inverse_fft** 3
- 11.1.1 Example 1: FFT Spectrum of a Monochromatic Signal 3
- 11.1.2 Example 2: FFT Spectrum of a Sum of Two Monochromatic Signals 8
- 11.1.3 Example 3: FFT Spectrum of a Rectangular Wave 10
- 11.1.4 Example 4: FFT Spectrum Sidebands of a Tone Burst Before and After Filtering 13
- 11.1.5 Example 5: Cleaning a Noisy Signal using FFT Methods 17

11.2 Our Notation for the Discrete Fourier Transform and its Inverse 22

11.3 Syntax of **qfft.mac** Functions 25

11.4 The Discrete Fourier Transform Derived via a Numerical Integral Approximation 27

11.5 Fast Fourier Transform References 28

*This is a live document. This version uses **Maxima 5.19.0**. Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions for improvements to `woollett@charter.net`

Preface

```
COPYING AND DISTRIBUTION POLICY
This document is part of a series of notes titled
"Maxima by Example" and is made available
via the author's webpage http://www.csulb.edu/~woollett/
to aid new users of the Maxima computer algebra system.

NON-PROFIT PRINTING AND DISTRIBUTION IS PERMITTED.

You may make copies of this document and distribute them
to others as long as you charge no more than the costs of printing.

These notes (with some modifications) will be published in book form
eventually via Lulu.com in an arrangement which will continue
to allow unlimited free download of the pdf files as well as the option
of ordering a low cost paperbound version of these notes.
```

Feedback from readers is the best way for this series of notes to become more helpful to new users of Maxima. *All* comments and suggestions for improvements will be appreciated and carefully considered.

```
LOADING FILES
The defaults allow you to use the brief version load(fft) to load in the
Maxima file fft.lisp.
To load in your own file, such as qfft.mac (used in this chapter),
using the brief version load(qfft), you either need to place
qfft.mac in one of the folders Maxima searches by default, or
else put a line like:

file_search_maxima : append(["c:/work3/###.{mac,mc}"],file_search_maxima )$

in your personal startup file maxima-init.mac (see Ch. 1, Introduction
to Maxima for more information about this).

Otherwise you need to provide a complete path in double quotes,
as in load("c:/work3/qfft.mac"),

We always use the brief load version in our examples, which are generated
using the Xmaxima graphics interface on a Windows XP computer, and copied
into a fancy verbatim environment in a latex file which uses the fancyvrb
and color packages.
```

```
  Maxima, a Computer Algebra System.
  Version 5.19.0 (2009) using Lisp GNU Common Lisp (GCL) GCL 2.6.8
  (aka GCL).  http://maxima.sourceforge.net/
```

The homemade function **fll(x)** (first, last, length) is used to return the first and last elements of lists (as well as the length), and is defined in the Ch.1 utility file **mbe1util.mac**. We will include a reference to this definition when working with lists.

This function is defined by:

```
fll(x) := [first(x),last(x),length(x)]$
declare(fll,evfun)$
```

The author would like to thank the Maxima developers for their friendly help via the Maxima mailing list.

11.1 Examples of the Use of the Fast Fourier Transform Functions fft and inverse_fft

We discuss the use of Maxima's fast Fourier transform package **fft.lisp**, which has been rewritten (effective with version 5.19) to allow list input and output (rather than being restricted to array input and output).

We first present five simple examples of using Maxima's fast Fourier transform functions **fft** and **inverse_fft**. We then discuss our notation and the utility functions available in the Ch.11 file **qfft.mac**. We then present a derivation of the discrete Fourier transformation pairs, using Maxima's conventions.

11.1.1 Example 1: FFT Spectrum of a Monochromatic Signal

We load in the Maxima package **fft.lisp** and also our own package **qfft.mac**.

```
(%i1) load(fft);
(%o1)  C:/PROGRA~1/MA89DF~1.0/share/maxima/5.19.0/share/numeric/fft.lisp
(%i2) load(qfft);
(%o2)                          c:/work3/qfft.mac
```

Our first example uses the simplest possible signal which still contains some information, a signal having one intrinsic frequency. We assume the signal is $\mathbf{F}(\mathbf{t}) = \cos(6\,\pi\,t)$. This signal has an angular frequency $\omega = 6\,\pi = 2\,\pi\,f$ where **f** is the frequency in Hertz (ie., in $\sec^{-1}$). Thus the frequency $\mathbf{f} = 3\sec^{-1}$. We bind the symbol **e** to this signal expression for later use:

```
(%i3) e : cos ( 6*%pi*t )$
```

Let **ns** be the (even) number of function samples, and **fs** be the sampling frequency.

The **qfft** package function **nyquist(ns, fs)** computes the time interval between signal samples **dt = 1/fs**, the Nyquist integer **knyq = ns/2**, the frequency resolution **df = fs/ns**, and the Nyquist frequency **fnyq = knyq * df = fs/2**.

The product of the time interval **dt** between samples of the signal and the frequency resolution **df** is always the inverse of the total number of signal samples **ns**:

```
dt * df = 1 / ns
```

We select **fs** and **ns** to satisfy two conditions. The only intrinsic frequency of the signal is $\mathrm{f0} = 3\,\mathrm{s}^{-1}$. **First** we need $\mathrm{fs} > 2\,\mathrm{f0}$ which means that $\mathrm{fs} > 6\,\mathrm{s}^{-1}$. **Secondly** we need the frequency resolution **df** to satisfy $\mathrm{df} < \mathrm{f0}$, which will be satisfied if we choose **df** such that $\mathrm{f0} = 3\,\mathrm{df}$, or **df = 1 hertz**. But then **df = fs / ns = 1**, or **fs = ns**, so the first condition becomes $\mathrm{ns} > 6$, but we also need **ns** to be an even number of the form 2^{m} for some integer m, so we choose **ns = 8**.

We then bind both **ns** and **fs** to the value **8** and assign **dt** (the sampling interval) to the first element of the list produced by the package function **nyquist(ns, fs)**, which also prints out **dt**, **knyq**, **fnyq**, and **df**.

```
(%i4) ( ns : 8, fs : 8 )$
(%i5) dt : first (nyquist (ns,fs));
sampling interval dt =  0.125
Nyquist integer knyq =  4
Nyquist freq fnyq =  4.0
freq resolution df =  1.0
(%o5)                               0.125
```

We then use the package function **sample(expr, var, ns, dvar)** to generate a list of floating point numbers as the expression **expr** is evaluated **ns** times, generating the list (with **dvar** being the time step **dt** here) **[F(0),F(dt),F(2*dt),...,F((ns-1)*dt)]** holding the values **F(m*dt)**, for **m = 0,1,2,...,ns-1**. More details of the syntax of the **qfft.mac** package functions can be found in Section 11.3. Recall that we have just bound **ns** and **dt** to numerical values and that the expression **e** depends on the variable **t**.

```
(%i6) flist : sample (e,t,ns,dt);
(%o6) [1.0, - 0.707, - 1.83691E-16, 0.707, - 1.0, 0.707, 5.51073E-16, - 0.707]
```

We see that elements three, **F(2*dt)**, and seven, **F(6*dt)**, are tiny numbers of the order of the default floating point errors, and are numerically equivalent to zero.

We first make a plot of **both** the given function (the signal) and a list of points **[m*dt, F(m*dt)]** constructed using the package function **vf (flist, dt)**. Recall that the signal is first sampled at **t = 0**.

```
(%i7) tflist : vf (flist,dt);
(%o7) [[0, 1.0], [0.125, - 0.707], [0.25, - 1.83691E-16], [0.375, 0.707],
          [0.5, - 1.0], [0.625, 0.707], [0.75, 5.51073E-16], [0.875, - 0.707]]
```

We let **tmax** be the “sampling time”, the number of signal samples times the time interval between samples.

```
(%i8) tmax : ns*dt;
(%o8)                                1.0
(%i9) plot2d([e ,[discrete,tflist]], [t,0,tmax],
             [style,[lines,3],[points,3,2,1]],
              [legend,false])$
```

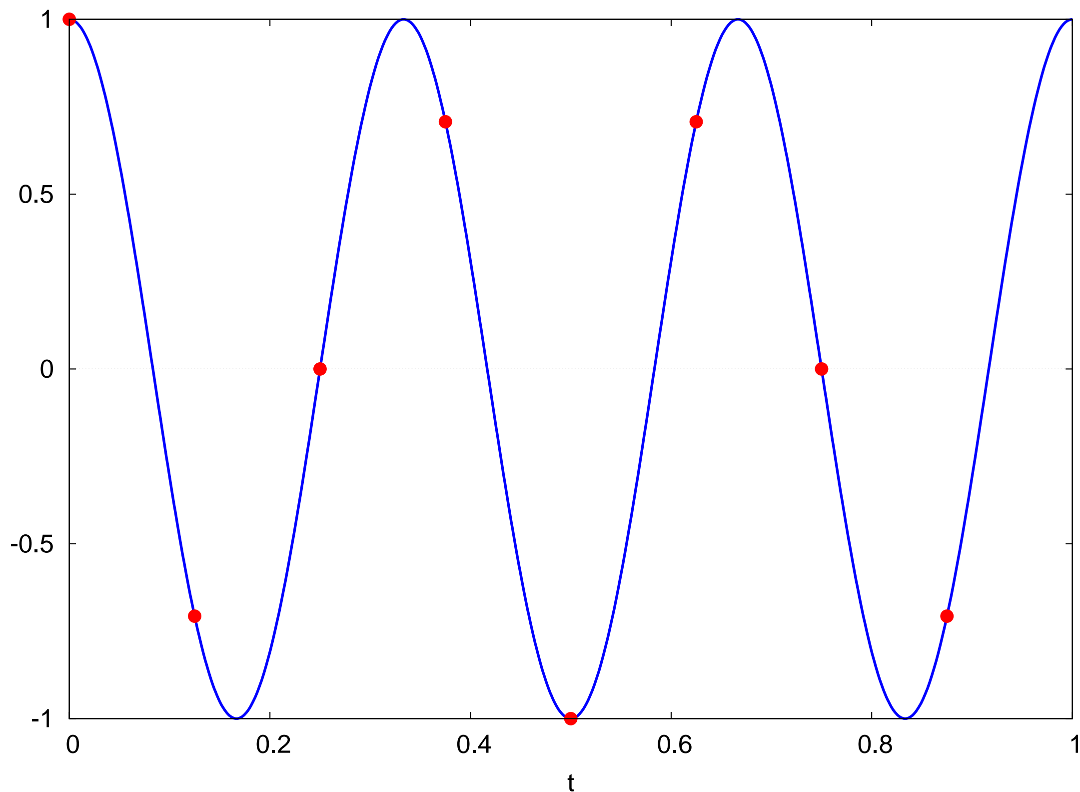

Figure 1: $F(t) = \cos(6\,\pi\,t)$ with Sample Points

Now that we have looked at both the signal and sample signal values (on the same plot), we next look at the **fast Fourier frequency spectrum** implied by this single frequency signal sample. We first define **glist** to be the fast fourier transform of **flist**

```
(%i10) glist : fft (flist);
(%o10) [- 2.34662E-17, 1.11022E-16 - 3.30546E-17 %i,
1.52656E-16 %i - 4.59227E-17, 0.5 - 1.18171E-16 %i, 1.15312E-16,
2.16746E-16 %i + 0.5, - 1.52656E-16 %i - 4.59227E-17,
5.55112E-17 - 6.55197E-17 %i]
(%i11) fchop (%);
(%o11)              [0.0, 0.0, 0.0, 0.5, 0.0, 0.5, 0.0, 0.0]
```

in which we have used Maxima's fast fourier transform function **fft**. We then used our package function **fchop** to set tiny floating point numbers to zero. The **qfft** package function **current_small()** prints out the current setting of the **qfft.mac** parameter **_small%** used by **fchop**.

```
(%i12) current_small()$
     current default small chop value =  1.0E-13
```

The first element of **glist** corresponds to **k = 0**, and the list **glist** contains the values of the fast Fourier transform amplitudes **G(k*df)** for **k = 0, 1, 2, ..., ns -1**, where **df** is the frequency resolution (for this example, **df = 1 hertz**). We call the value **G(k*df)** the "fast Fourier transform amplitude" corresponding to the frequency **f = k*df**.

If we look at the chopped version, we see that the first non-zero fast fourier transform amplitude occurs at element four, which corresponds to **k = 3**, which corresponds to the frequency **f = k*df = 3 * 1 = 3 hertz**. In a later section exploring the basic ideas of the fast Fourier transform, we explain why all the usable spectrum information is contained in the interval **k = 0** to **k = knyq = 4**.

To make a simple point plot of the fourier amplitudes, we first use the package function **kg** to make a list of the points **[k, G(k*df)]** out to the Nyquist integer **knyq**. Since we want real numbers for a plot, this function takes the absolute value of each Fourier amplitude and also chops tiny numbers.

```
(%i13) kglist : kg (glist);
(%o13)        [[0, 0.0], [1, 0.0], [2, 0.0], [3, 0.5], [4, 0.0]]
(%i14) plot2d ( [discrete, kglist],[x,0,5],[y,0,0.8],
                  [style,[points,5,1,1]],[xlabel,"k"],
                  [ylabel," "],[gnuplot_preamble,"set grid;"])$
```

which produces the simple plot

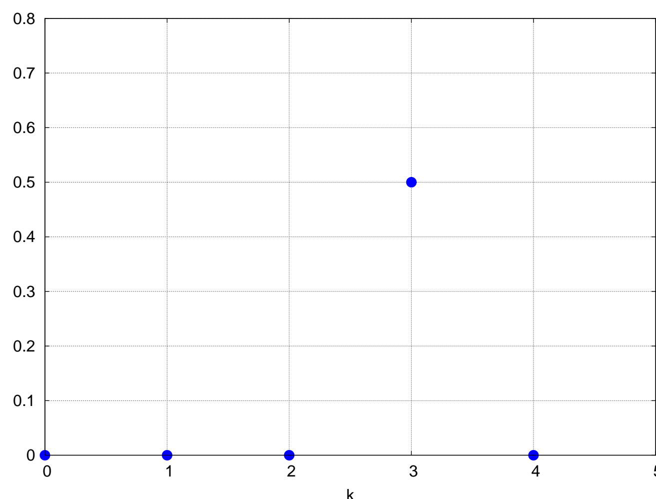

Figure 2: Spectrum of $\cos(6\,\pi\,t)$

We can also use **plot2d** to make a simple histogram, using **kglist**.

```
(%i15) vbars : makelist ( [discrete,
                  [[kglist[i][1],0],[kglist[i][1],kglist[i][2]]]] ,
                      i,1,length(kglist) );
(%o15) [[discrete, [[0, 0], [0, 0.0]]], [discrete, [[1, 0], [1, 0.0]]],
[discrete, [[2, 0], [2, 0.0]]], [discrete, [[3, 0], [3, 0.5]]],
[discrete, [[4, 0], [4, 0.0]]]]
(%i16) plot2d ( vbars,[y,0,0.8],[style,[lines,5,1]],
               [ylabel," "],[xlabel," k "],[legend,false],
                [gnuplot_preamble,"set grid;"] )$
```

which produces the plot

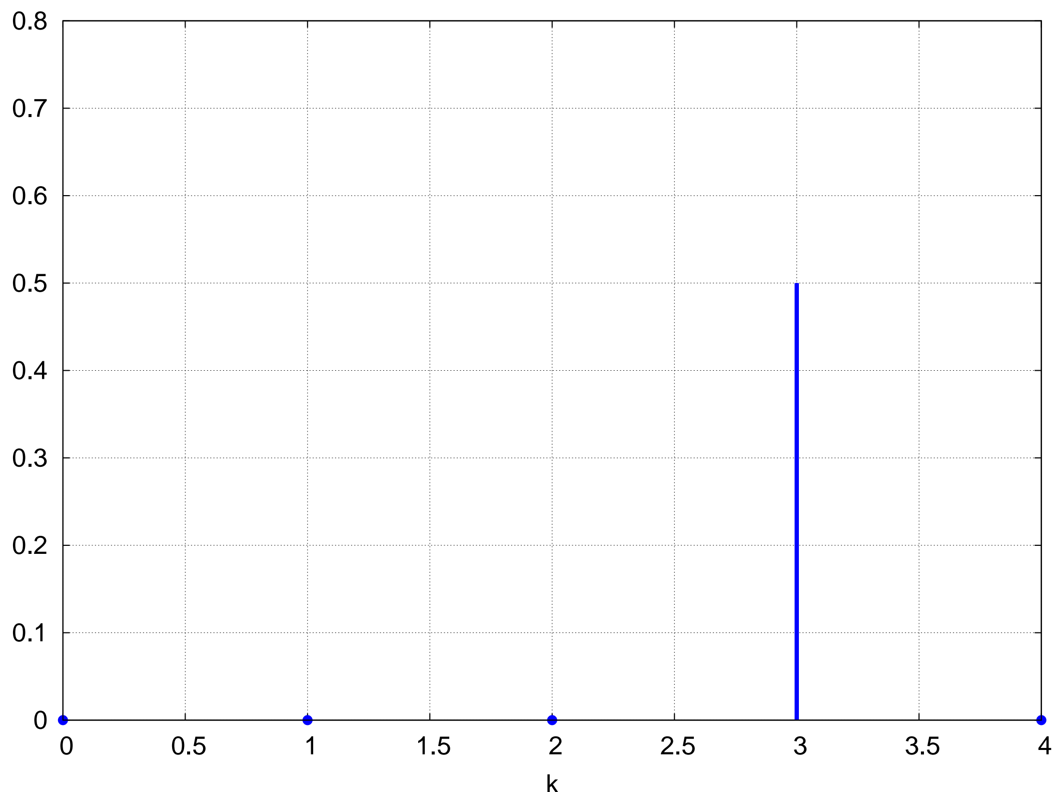

Figure 3: Line Spectrum of $\cos(6\,\pi\,t)$

The production of such a frequency space spectrum plot can be automated by accepting the fast Fourier transform list **glist** and defining a Maxima function **spectrum (glist, nlw, ymax)** which would be used with the specific syntax **spectrum (glist, 5, 0.8)** to obtain a histogram similar to the above but cleaner. (**nlw** is the line width and **ymax** is the vertical extent of the canvas starting at `y = 0`.)

We have designed **spectrum** to also allow the expanded syntax **spectrum (glist, nlw, ymax, k1, k2)** which zooms in on the interval `k1 <= k <= k2`, where `k2 <= knyq`. In addition, we have avoided, in **spectrum**, creating vertical bars when a fast Fourier transform amplitude is less than a very small number.

The entry

```
(%i17) spectrum (glist, 5, 0.8 )$
```

produces the plot

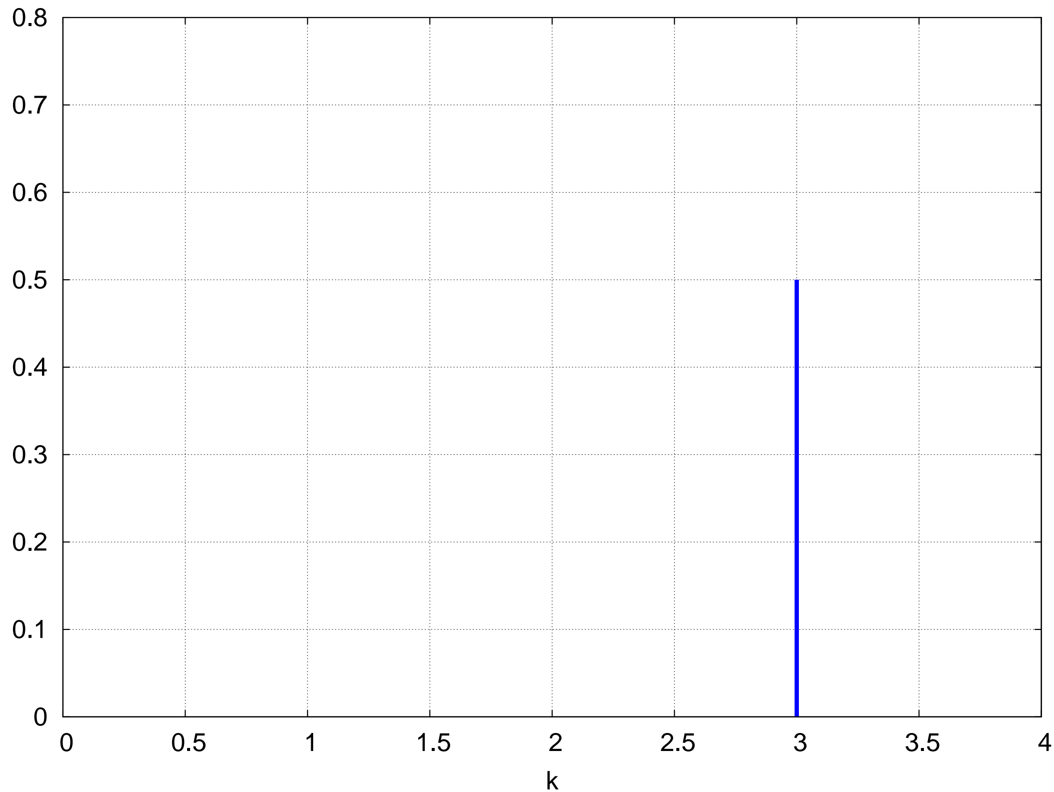

Figure 4: spectrum (glist, 5, 0.8)

We will use **spectrum**, included in **qfft.mac**, in our following examples as a quick way to look at the frequency spectrum.

Finally, let's use the Maxima function **inverse_fft** to apply the inverse fast Fourier transform to the list of fast Fourier transform amplitudes **glist**, and see how closely the result matches the original signal sample list **flist**.

```
(%i18) flist1 : inverse_fft(glist);
(%o18) [1.0, 1.11022E-16 %i - 0.707, 2.24949E-32 %i - 1.83691E-16,
6.77259E-17 %i + 0.707, - 1.0, 0.707 - 1.11022E-16 %i,
5.51073E-16 - 2.24949E-32 %i, - 6.77259E-17 %i - 0.707]
(%i19) fchop(%);
(%o19)      [1.0, - 0.707, 0.0, 0.707, - 1.0, 0.707, 0.0, - 0.707]
(%i20) fchop( flist);
(%o20)      [1.0, - 0.707, 0.0, 0.707, - 1.0, 0.707, 0.0, - 0.707]
(%i21) lmax ( abs (flist1 - flist));
(%o21)                          1.30049E-16
```

We see that **inverse_fft (glist)** recovers our original signal sample list **flist** to within floating point errors.

11.1.2 Example 2: FFT Spectrum of a Sum of Two Monochromatic Signals

Almost all signals will contain more than one intrinsic frequency, and to recover a portion of the frequency spectrum with fidelity, we need to satisfy the two conditions

$$\mathrm{fs} > 2\,\mathrm{f_{high}} \quad \text{and} \quad \mathrm{df} \le \mathrm{f_{low}}, \tag{11.1}$$

in which $\mathrm{f_{low}}$ is the lowest intrinsic frequency to be identified, and $\mathrm{f_{high}}$ is the highest intrinsic frequency to be identified. (Again, **fs** is the sampling frequency, and **df** is the frequency resolution).

We assume now that the signal is $\mathrm{F(t)} = \cos(2\,\pi\,t) + \sin(4\,\pi\,t)$. We thus have a signal with the frequencies $\mathrm{f_1} = 1\,\mathrm{s}^{-1}$ and $\mathrm{f_2} = 2\,\mathrm{s}^{-1}$. With fs being the sampling frequency, and ns being the number of signal samples, we require $\mathrm{fs} > 2\,\mathrm{f_{high}} = 4$, as well as $\mathrm{df} \le \mathrm{f_{low}} = 1$. If we choose $\mathrm{f_{low}} = 3\,\mathrm{df}$, then `df = 1/3 = fs/ns`, or `fs = ns/3`. So the first condtion then implies we need $\mathrm{ns}/3 > 4$, or $\mathrm{ns} > 12$. Since `ns` also should be equal to 2^{m} for some integer m, we choose `ns = 16`. Then `fs = 16/3` and `df = fs/ns = 1/3 hertz`.

```
(%i1) ( load(fft), load(qfft) )$
(%i2) e : cos(2*%pi*t) + sin(4*%pi*t)$
(%i3) (ns : 16, fs : 16/3)$
(%i4) dt : first (nyquist (ns,fs));
sampling interval dt =  0.188
Nyquist integer knyq =  8
Nyquist freq fnyq =  2.6667
freq resolution df =  0.333
(%o4)                                   0.188
(%i5) flist : sample (e,t,ns,dt)$
(%i6) %,fll;
(%o6)                          [1.0, - 0.324, 16]
(%i7) tmax: ns*dt;
(%o7)                                    3.0
(%i8) tflist : vf (flist,dt)$
(%i9) %,fll;
(%o9)                   [[0, 1.0], [2.8125, - 0.324], 16]
(%i10) plot2d([e ,[discrete,tflist]], [t,0,tmax],
             [style,[lines,3],[points,3,2,1]],
              [legend,false])$
```

We have used our utility function **fll** described in the preface, which returns the first and last element of a list, as well as the length of the list. The plot thus produced is

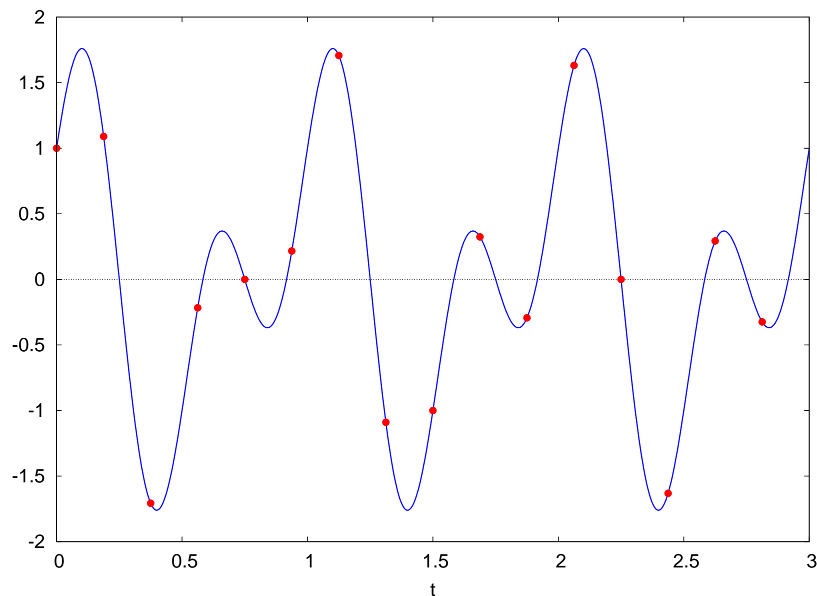

Figure 5: $\cos(2\,\pi\,t) + \sin(4\,\pi\,t)$ with Sample Points

We next generate the list **glist** of fast Fourier transform amplitudes **`G(k*df)`** and use **spectrum** to look at the implied frequency spectrum.

```
(%i11) glist : fft (flist)$
(%i12) %,fll;
(%o12)          [- 7.94822E-17, 1.52656E-16 - 5.7151E-17 %i, 16]
(%i13) spectrum (glist,5,0.6)$
```

which produces the spectrum histogram:

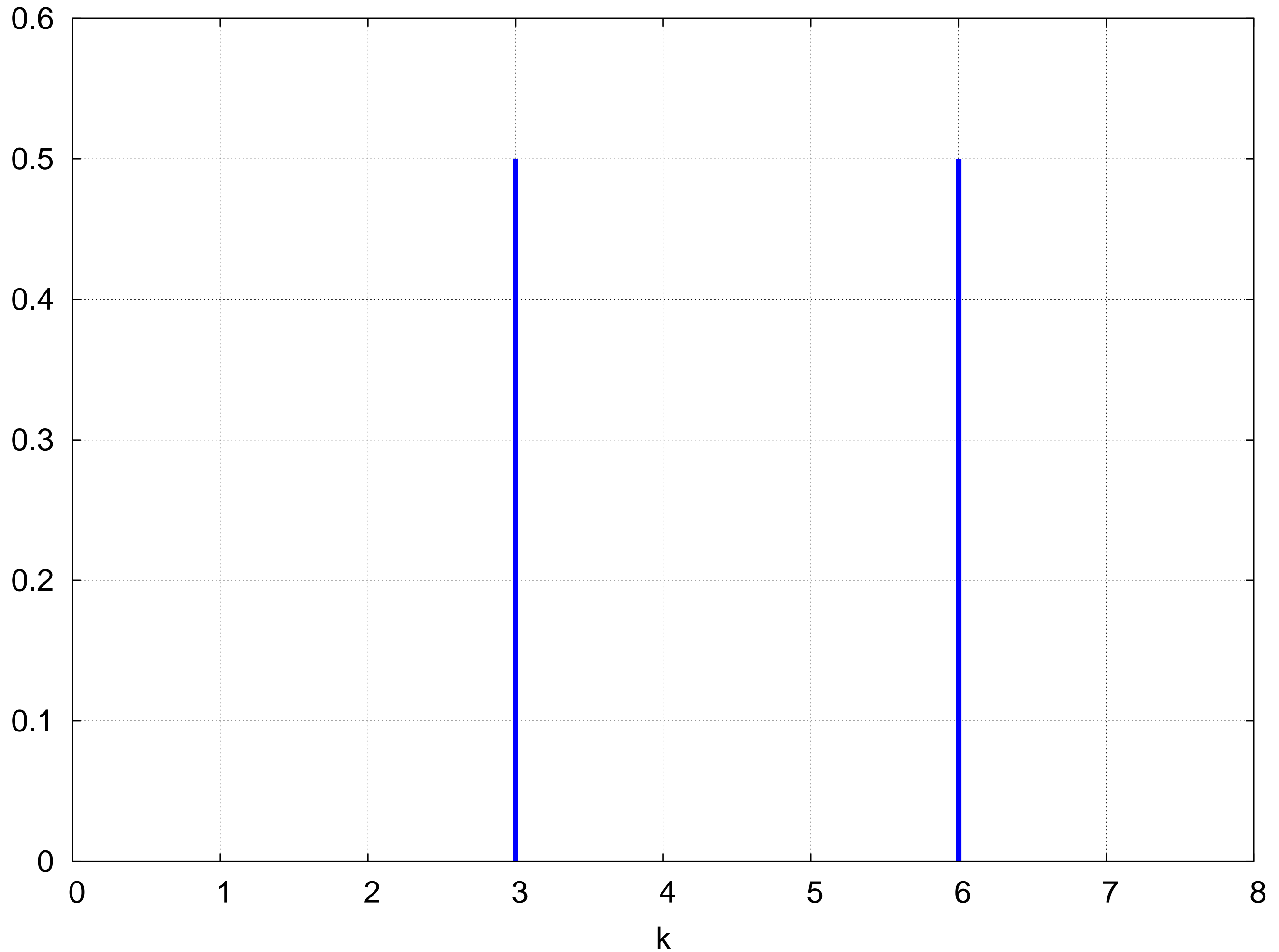

Figure 6: Spectrum of $\cos(2\,\pi\,t) + \sin(4\,\pi\,t)$

Since **`df = 1/3`**, the line at **`k = 3`** corresponds to the frequency **`f = k*df = 3*(1/3) = 1 Hertz`** and the line at **`k = 6`** corresponds to the frequency **`f = k*df = 6*(1/3) = 2 Hertz`**.

11.1.3 Example 3: FFT Spectrum of a Rectangular Wave

A rectangular wave as a function of time **t** with period equal to **64 sec** which cycles between plus and minus **1** can be constructed from **floor** and **mod** in the form: **rwave(t) := 2*mod(floor(t/32),2) - 1**.

For **t = 0, t/32 = 0, floor(t/32) = floor(0) = 0, mod(0,2) = 0**, so **rwave(0) = -1**.

```
(%i1) rwave(t):= 2*mod(floor(t/32),2) -1 $
(%i2) [floor(0),mod(0,2),rwave(0)];
(%o2)                              [0, 0, - 1]
```

For **t = 32, floor(32/32) = floor(1) = 1, mod(1,2) = 1, rwave(32) = 1**.

```
(%i3) [floor(1),mod(1,2),rwave(32)];
(%o3)                               [1, 1, 1]
```

For **t = 64, floor(64/32) = floor(2) = 2, mod(2,2) = 0, rwave(64) = -1**.

```
(%i4) [floor(2),mod(2,2),rwave(64)];
(%o4)                              [2, 0, - 1]
```

Hence **rwave(0 + 64) = rwave(0)** and **rwave(t)** has a period equal to **64 sec**.

We first look at the rectangular wave signal with a plot.

```
(%i5) plot2d(rwave(t),[t,0,128],[y,-1.5,1.5],
         [ylabel," "],[style,[lines,5]],
         [gnuplot_preamble,"set grid;set zeroaxis lw 2;"])$
```

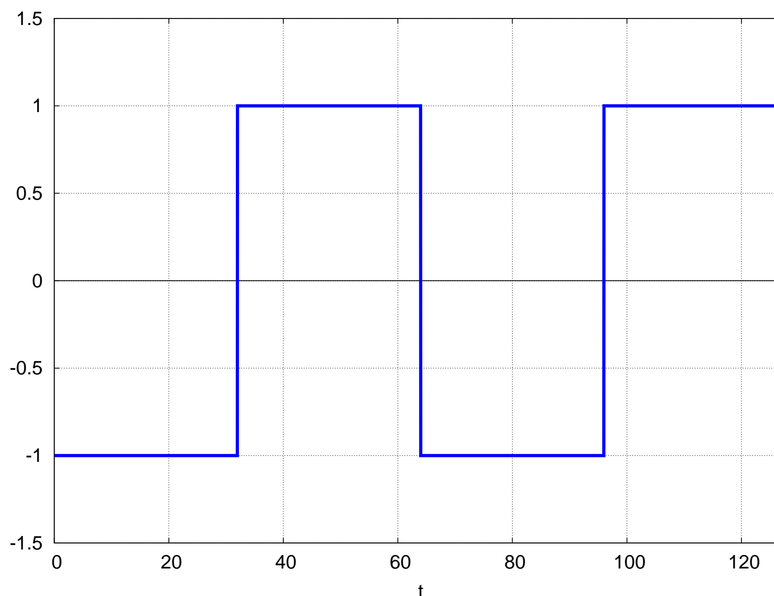

Figure 7: Rectangular Wave with Period 64 sec.

If we consider sampling this signal at intervals **dt = 1 sec**, the sampling frequency will be **fs = 1/dt = 1 hertz = 1 per sec = 64 per cycle**. The lowest intrinsic frequency of this signal is **f_low =** 1/64 hertz—, corresponding to the period of the rectangular wave. If we choose **4*df = f_low**, then **df = 1/256 = fs/ns = 1/ns**, so **ns = 256**. Due to the sharp corners of a rectangular wave we expect many high frequency components to be present in the spectrum.

We thus try the combination, **ns = 256**, and **fs = 1 hertz**.

```
(%i6) (ns:256, fs:1)$
(%i7) (load(fft),load(qfft) )$
(%i8) dt:first(nyquist(ns,fs));
sampling interval dt =  1.0
Nyquist integer knyq =  128
Nyquist freq fnyq =  0.5
freq resolution df =  0.00391
(%o8)                              1.0
(%i9) flist : sample(rwave(t),t,ns,dt)$
(%i10) fll (flist);
(%o10)                        [- 1.0, 1.0, 256]
(%i11) makelist (flist[i],i,1,10);
(%o11) [- 1.0, - 1.0, - 1.0, - 1.0, - 1.0, - 1.0, - 1.0, - 1.0, - 1.0, - 1.0]
(%i12) tmax: ns*dt;
(%o12)                             256.0
(%i13) tflist : vf (flist,dt)$
(%i14) fll (tflist);
(%o14)                 [[0, - 1.0], [255.0, 1.0], 256]
(%i15) plot2d([rwave(t) ,[discrete,tflist]], [t,0,tmax],
              [y,-1.5,1.5],[ylabel," "],
             [style,[lines,3],[points,1,0,1]],
              [legend,false])$
```

The **sample_plot** invocation produces the black points of the sample on top and bottom of the rectangular wave shown here:

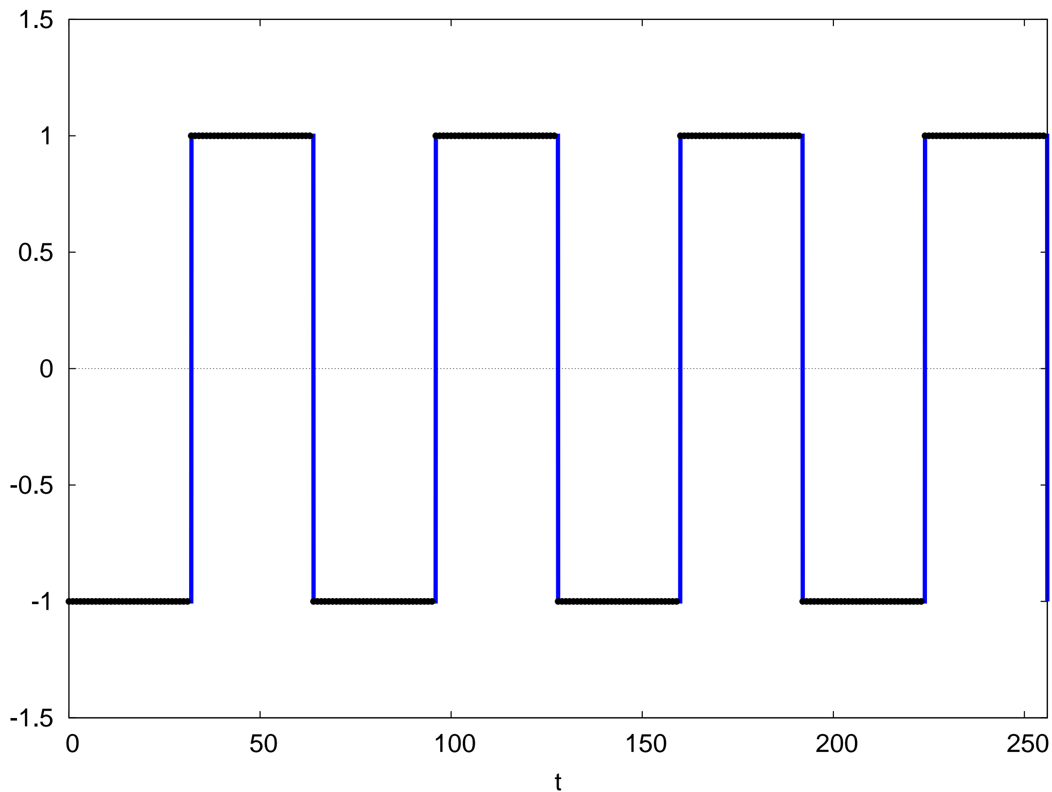

Figure 8: Black Sample Points with Rectangular Wave

Next we look at the signal frequency spectrum from **k = 0** to **k = knyq = 128**. Since **f = k*df**, and **df = 1/256 hertz**, the low fundamental intrinsic frequency **f_0 = 1/64 hertz** will be located at **k = 4**, and the maximum possible frequency component will be **fnyq = knyq*df = 0.5 hertz = 32*f_0**. The spectrum plot is generated by passing the list **flist** containing **ns = 256** signal samples, along with **nlw = 3**, and **ymax = 0.7** to **spectrum**:

```
(%i16) glist : fft (flist)$
(%i17) %,fll;
(%o17)                          [0.0, 0.0, 256]
(%i18) spectrum (glist,3,0.7)$
```

which produces the spectrum plot:

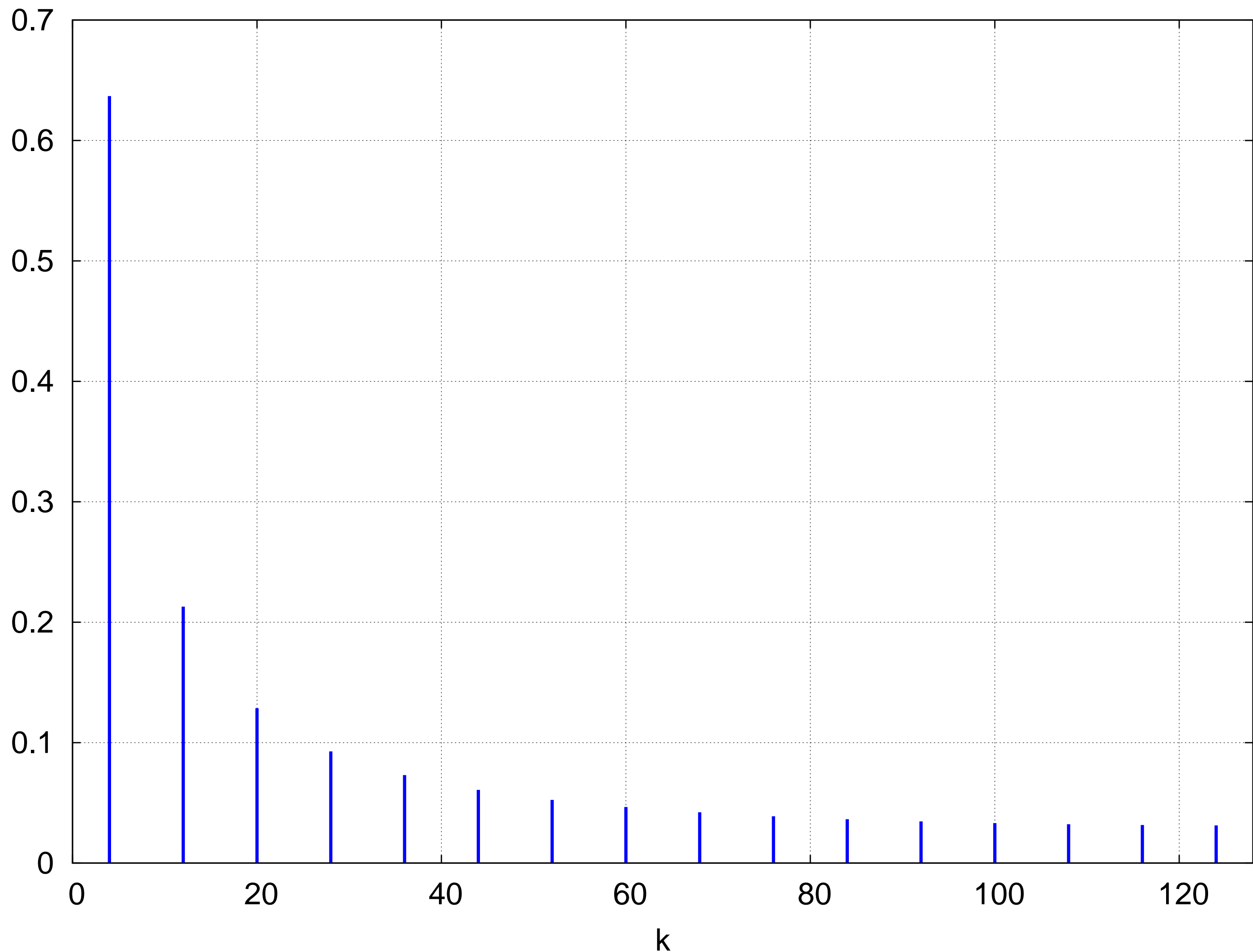

Figure 9: Spectrum from k = 0 to knyq = 128

The plot shows lines at **k = 4, 12, 20, 28, 36, ...** Since **f = k*df** and **f_0 = 4*df** is the fundamental frequency, the frequencies present are **f_0, 3*f_0, 5*f_0, 7*f_0,...** which is the "fundamental" plus odd harmonics of the fundamental. We know there must be high frequency components to account for the sharp corners of the signal.

11.1.4 Example 4: FFT Spectrum Sidebands of a Tone Burst Before and After Filtering

A tone burst signal consisting of a sine wave begins at `t = 0` and ends abruptly after forty five seconds. The fast Fourier frequency spectrum of such a signal, when sampled over the duration of the sine wave plus some time following the sine wave end, will contain "sideband" frequencies above and below the intrinsic frequency of the sine wave. Many signal filters can be used to allow the easy identification of the intrinsic sine wave frequency by surpressing the sideband frequencies. We use for the tone burst the sine wave: $\sin(2\,\pi\,t/5)$ during the time interval $0 \le t \le 45\,\mathrm{sec}$, and then abruptly dropping to `0` for $t > 45\,\mathrm{sec}$. The intrinsic frequency of the sine wave is `f0 = 1/5 hertz`, and the corresponding period is `5 sec`, thus the sine wave lasts for nine periods. We will use the following idealized model of such a signal by ignoring the actual time needed to end the sine wave.

```
(%i1) sig(t) := if t < 45 then sin(2*%pi*t/5.0)  else 0$
```

We will use this definition only for $t \ge 0$ in our work. With `fs` the sampling frequency, if we want about **10 samples/cycle** (ie., `10 per 5 sec`), then we want `period*fs = 10`, or `fs` roughly equal to `2 hertz`. We want the intrinsic, frequency `f0` (`0.2 hertz`) to not be too close to the left end of the frequency spectrum plot so we can see the low frequency sidebands as well as the higher frequency sidebands.

Suppose we try requiring that `f0 = 0.2 hertz = 50*df = 50*fs/ns`. Solving for the number of signal samples `ns`, we get `ns = 50 * 2 * 5 = 500`. To get a power of **2** we choose the closest such quantity, `ns = 512`. We then return to our `f0` location requirement to solve for the resulting value of `fs` to get **fs = 512/250 = 2.048 hertz**.

```
(%i2) load (qfft)$
(%i3) (ns:512,fs:2.048)$
(%i4) dt : first(nyquist(ns,fs));
sampling interval dt =  0.488
Nyquist integer knyq =  256
Nyquist freq fnyq =  1.024
freq resolution df =  0.004
(%o4)                              0.488
(%i5) tmax : ns*dt;
(%o5)                              250.0
```

We can plot a Maxima function defined with the **if then else** construct without problem since **plot2d** evaluates its arguments.

```
(%i6) plot2d ( sig(t),[t,0,tmax],[y,-1.5,1.5],
         [style,[lines,1,0]],[ylabel," "],[nticks,100],
          [legend,false],[gnuplot_preamble,"set grid;"])$
```

which produces the plot:

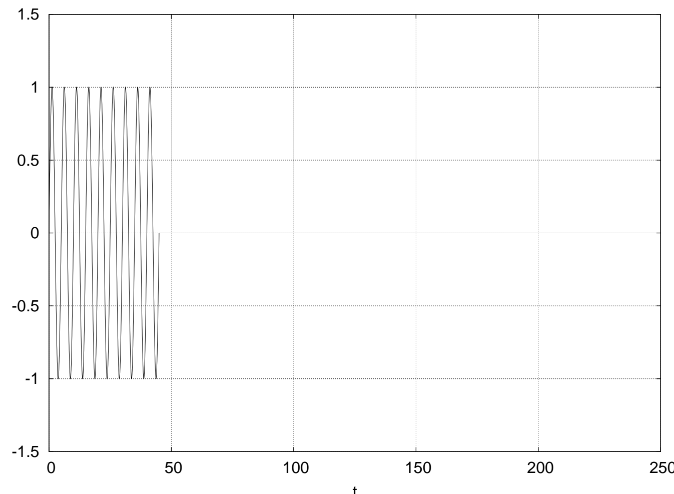

Figure 10: Sine Wave Tone Burst, Period = 5 sec, Duration = Nine Periods

Using the **qfft** package function **sample** with a Maxima function defined with the **if...then** construct produces a list each element of which includes **if then else**, which can be used with **plot2d**. However, we will include an extra evaluation in defining our **flist** so we can see the numerical values as a list. Note that the syntax **expr, fll;** causes an extra evaluation, so to be careful we need to use the syntax **fll (expr)**.

```
(%i7) flist : sample (sig(t),t,ns,dt)$
(%i8) fll (flist);
(%o8) [if 0.0 < 45.0 then 0.0 else 0.0,
                              if 249.51 < 45.0 then - 0.576 else 0.0, 512]
(%i9) flist : ev (flist)$
(%i10) fll (flist);
(%o10)                      [0.0, 0.0, 512]
(%i11) makelist(flist[i],i,1,10);
(%o11) [0.0, 0.576, 0.942, 0.964, 0.634, 0.0736, - 0.514, - 0.914, - 0.981,
                                                                    - 0.69]
(%i12) tflist : vf (flist,dt)$
(%i13) fll (tflist);
(%o13)                 [[0, 0.0], [249.51, 0.0], 512]
```

We now use **plot2d** to show our sample points on top of our tone burst signal (both in black).

```
(%i14) plot2d([sig(t) ,[discrete,tflist]], [t,0,tmax],
              [y,-1.5,1.5],[ylabel," "],
             [style,[lines,1,0],[points,1,0,1]],
              [legend,false])$
```

which produces the plot:

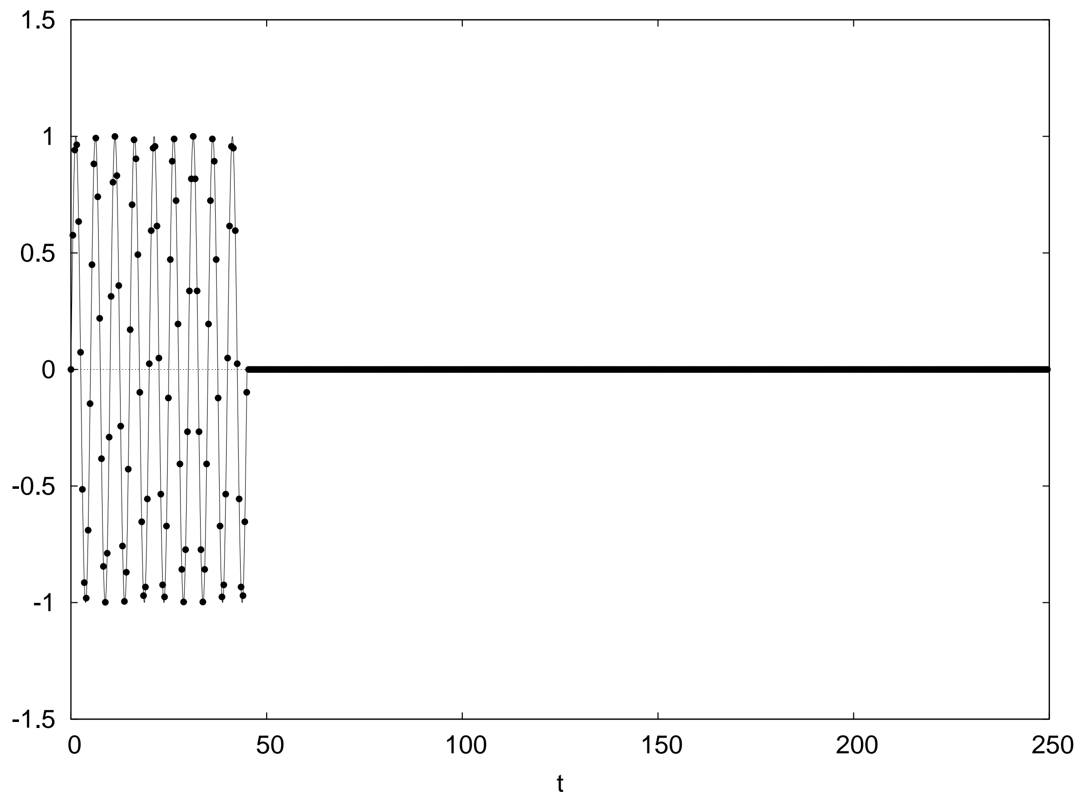

Figure 11: Signal and Sample Points Which Include 0's for $t > 45$ sec

We now plot the unfiltered tone burst spectrum after generating **glist** with **fft**.

```
(%i15) load (fft)$
(%i16) glist : fft (flist)$
(%i17) fll (glist);
(%o17)              [- 8.08763E-5, 0.00174 - 0.00286 %i, 512]
(%i18) spectrum (glist, 2, 0.1 )$
```

which produces the spectrum plot: We see strong sideband frequencies on both the low and high side of the intrinsic

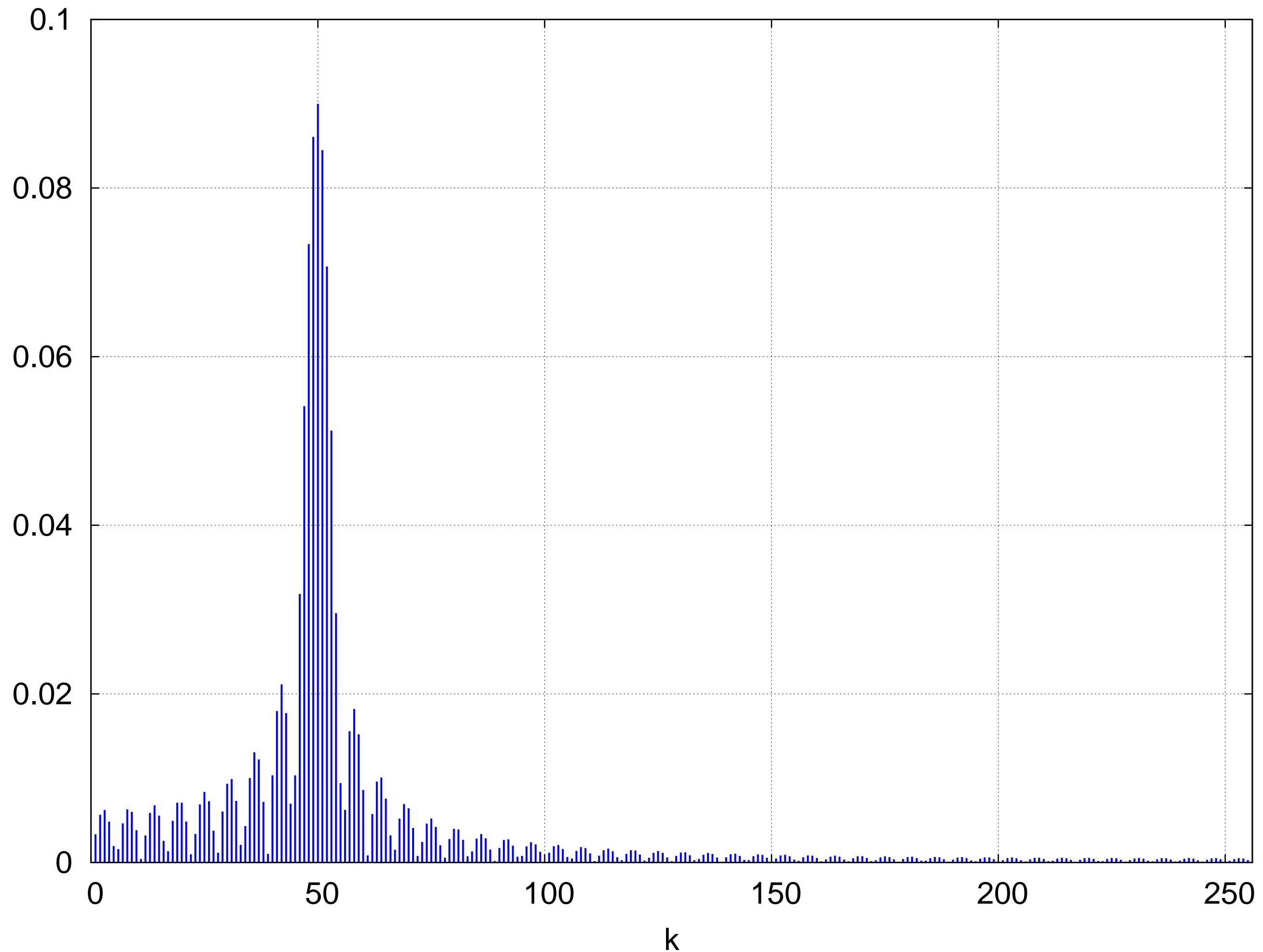

Figure 12: Frequency Spectrum of Unfiltered Tone Burst

frequency `f0 = 50*df`, which is the highest peak at `k = 50` (**f = k*df = 50*(1/250) = 1/5 = 0.02**). The sidebands are a mathematical artifact of the finite duration of the sine wave tone burst with its abrupt beginning and end.

A windowing filter which smooths out the beginning and end of the "burst envelope" is used by defining a smoothed envelope signal as the product of the unsmoothed signal and a suitable windowing function. The von Hann window employs a $\sin^2$ pinch envelope.

```
(%i19) hannw(x,m) := sin(%pi*(x-1)/(m-1))^2$
(%i20) sig_w(t) := hannw(t,45)*sig(t)$
(%i21) plot2d ( sig_w(t),[t,0,tmax],[y,-1.5,1.5],
         [style,[lines,1,0]],[ylabel," "],[nticks,100],
          [legend,false],[gnuplot_preamble,"set grid;"])$
```

which shows the filtered tone burst

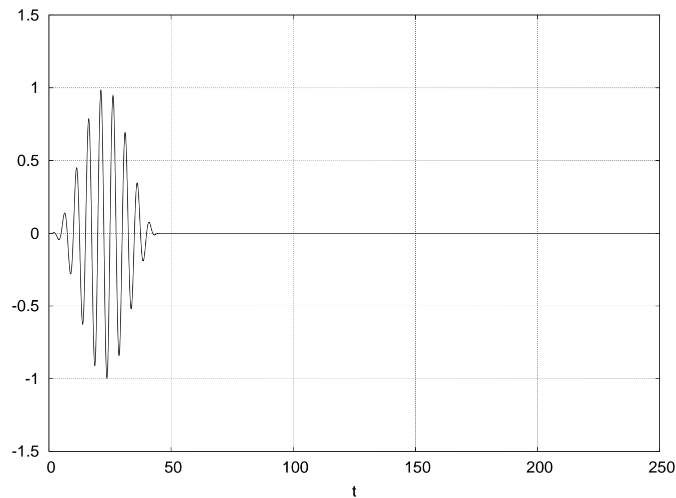

Figure 13: Filtered Tone Burst

Using now the filtered tone burst signal **sig_w(t)**, we construct a filtered signal sample list **flist_w** and look at the altered fast Fourier spectrum (remember **dt** has been bound to **0.488** and **ns** to **512**):

```
(%i22) flist_w : sample (sig_w(t),t,ns,dt)$
(%i23) fll (flist_w);
(%o23) [0.00509 (if 0.0 < 45.0 then 0.0 else 0.0),
                        0.799 (if 249.51 < 45.0 then - 0.576 else 0.0), 512]
(%i24) flist_w : ev (flist_w)$
(%i25) fll (flist_w);
(%o25)                        [0.0, 0.0, 512]
(%i26) makelist (flist_w[i],i,1,5);
(%o26)          [0.0, 7.68319E-4, 2.63668E-6, 0.00106, 0.00293]
(%i27) glist_w : fft (flist_w)$
(%i28) makelist (glist_w[i],i,1,5);
(%o28) [1.59068E-5, 3.93191E-6 - 1.99372E-5 %i, - 1.83151E-5 %i - 1.84837E-5,
                    2.09989E-6 %i - 2.59175E-5, 1.69797E-5 %i - 9.67992E-6]
(%i29) spectrum (glist_w, 2, 0.1)$
```

and we see deletion of most of the sideband frequency peaks:

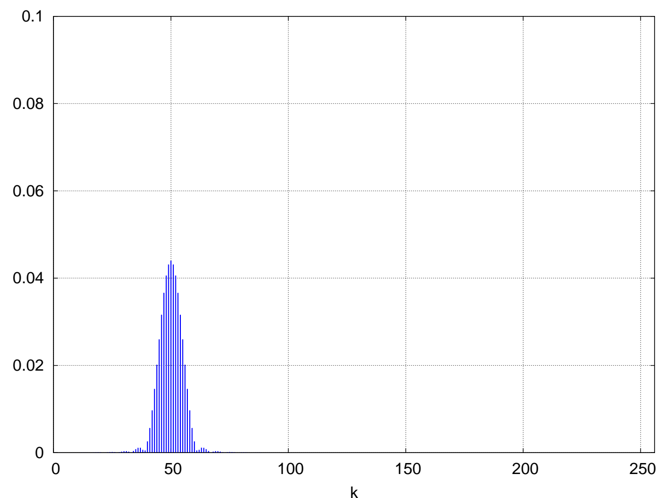

Figure 14: Frequency Spectrum of the Filtered Tone Burst

11.1.5 Example 5: Cleaning a Noisy Signal using FFT Methods

We use the same signal as used in Example 2, Sec. (11.1.2), but add some random noise to it. Without the noise, the signal is $\mathbf{F(t) = \cos(2\,\pi\,t) + \sin(4\,\pi\,t)}$. Thus the clean signal contains the frequencies $\mathbf{f_1 = 1\,s^{-1}}$ and $\mathbf{f_2 = 2\,s^{-1}}$, and two corresponding periods $\boldsymbol{\tau_1 = 1/f_1 = 1}\,\mathbf{sec}$ and $\boldsymbol{\tau_2 = 1/f_2 = 0.5}\,\mathbf{sec}$. To get a noisy looking signal we have to sample and add noise a lot of times within one or two periods of the clean signal, where we use the maximum of the intrinsic periods. Let's try $\mathbf{N = 512}$ samples over a time **tmax = 2 sec**. Then $\mathbf{N\,\delta t = 2\,sec}$, so **fs = 1/dt = N/2 = 256**. The first condition on the sampling frequency is that $\mathbf{f_s > 2\,f_{high}}$, or $\mathbf{fs > 4\,s^{-1}}$, which is certainly satisfied. The second condition on **fs** is $\mathbf{\delta f < f_{low}}$, or $\mathbf{df < 1\,s^{-1}}$, or $\mathbf{fs/ns < 1\,s^{-1}}$, or $\mathbf{fs < ns}$, which is also true. Hence we try **ns = 512, fs = 256**.

```
(%i1) e : cos(2*%pi*t) + sin(4*%pi*t)$
(%i2) (load(fft), load(qfft))$
(%i3) [ns:512,fs:256]$
(%i4) dt : first(nyquist(ns,fs));
sampling interval dt =  0.00391
Nyquist integer knyq =  256
Nyquist freq fnyq =  128.0
freq resolution df =  0.5
(%o4)                                   0.00391
(%i5) flist : sample(e,t,ns,dt)$
(%i6) %,fll;
(%o6)                              [1.0, 0.951, 512]
(%i7) flist_noise : makelist(flist[j]+0.3*(-1.0+random(2.0)),j,1,ns)$
(%i8) %,fll;
(%o8)                            [1.2483, 1.0073, 512]
(%i9) tflist_noise : vf (flist_noise,dt)$
(%i10) %,fll;
(%o10)                [[0, 1.2483], [1.9961, 1.0073], 512]
(%i11) plot2d ([discrete,tflist_noise],[y,-2,2],
                 [style,[lines,1]],[ylabel," "])$
```

which produces the noisy signal plot

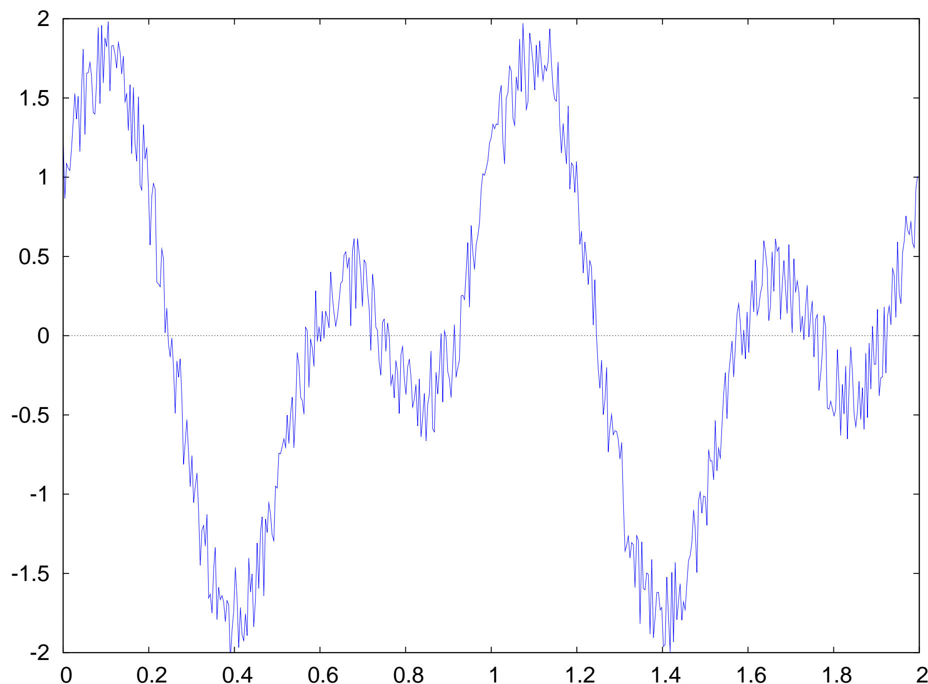

Figure 15: Noisy Signal

One way to "clean" this signal is to set small numbers in the fast Fourier transform to zero and inverse transform back to the time domain using **inverse_fft**.

In order to "chop" small numbers in the fast Fourier transform amplitude list, we need to produce that list, which we call **glist_noise** from the noisy signal sample **flist_noise** by using **fft**.

```
(%i12) glist_noise : fft (flist_noise)$
(%i13) %,fll;
(%o13)              [- 0.0022, 5.09918E-4 %i - 0.00338, 512]
```

Before "chopping" small numbers in **glist_noise**, we take a look at the fast Fourier frequency spectrum implied by **glist_noise**.

```
(%i14) spectrum (glist_noise,2,0.6)$
```

which produces the plot

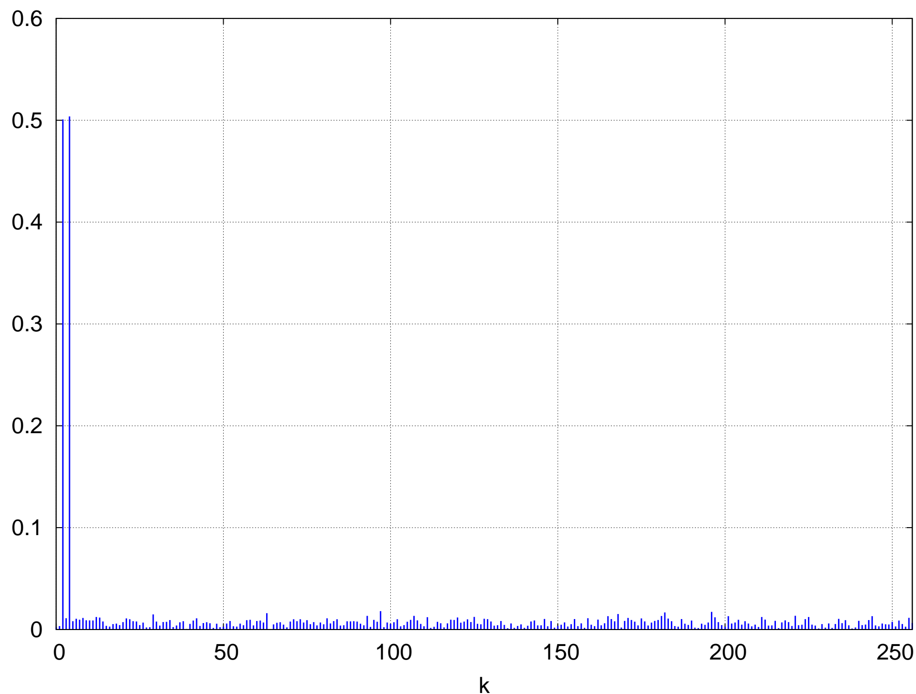

Figure 16: Noisy Signal Frequency Spectrum

The dominant lines are still those corresponding to the two intrinsic frequencies of the clean signal we started with, but there are many more frequencies present due to the noise.

To see more clearly the dominant line region, we show another plot with the integer **k** in the range **(0, 10)**:

```
(%i15) spectrum (glist_noise,4,0.6,0,10)$
```

which produces the plot:

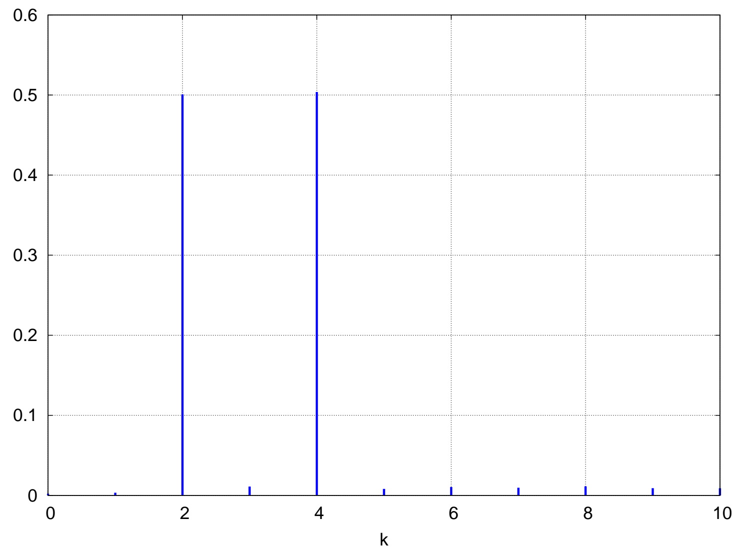

Figure 17: Noisy Signal Frequency Spectrum with k in (0,10)

We now use **fchop1** with the value **0.2** to set small floating point numbers less than **0.2** to zero in the frequency space list **glist_noise**, and again use **spectrum** to look at the frequency spectrum associated with the chopped frequency space list.

```
(%i16) glist_noise_chop : fchop1(glist_noise,0.2)$
(%i17) %,fll;
(%o17)                          [0.0, 0.0, 512]
(%i18) spectrum (glist_noise_chop,2,0.6)$
```

which produces the plot

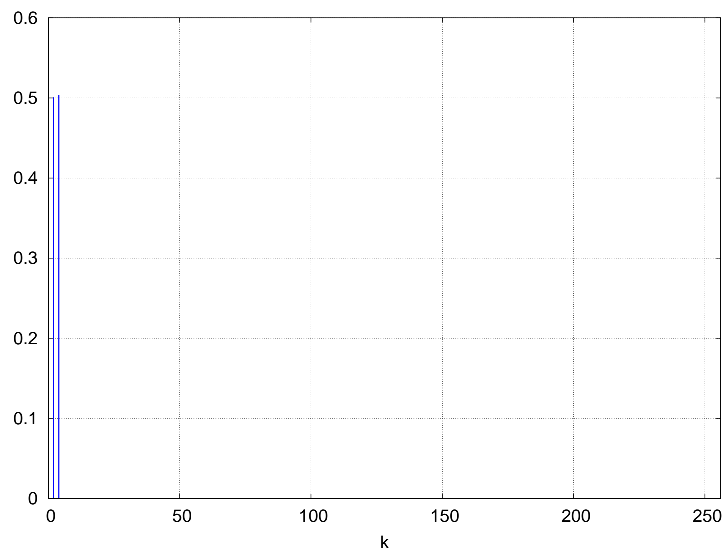

Figure 18: Chopped Frequency Spectrum

Here is the cleaned spectrum in the range **k = (0,10)**:

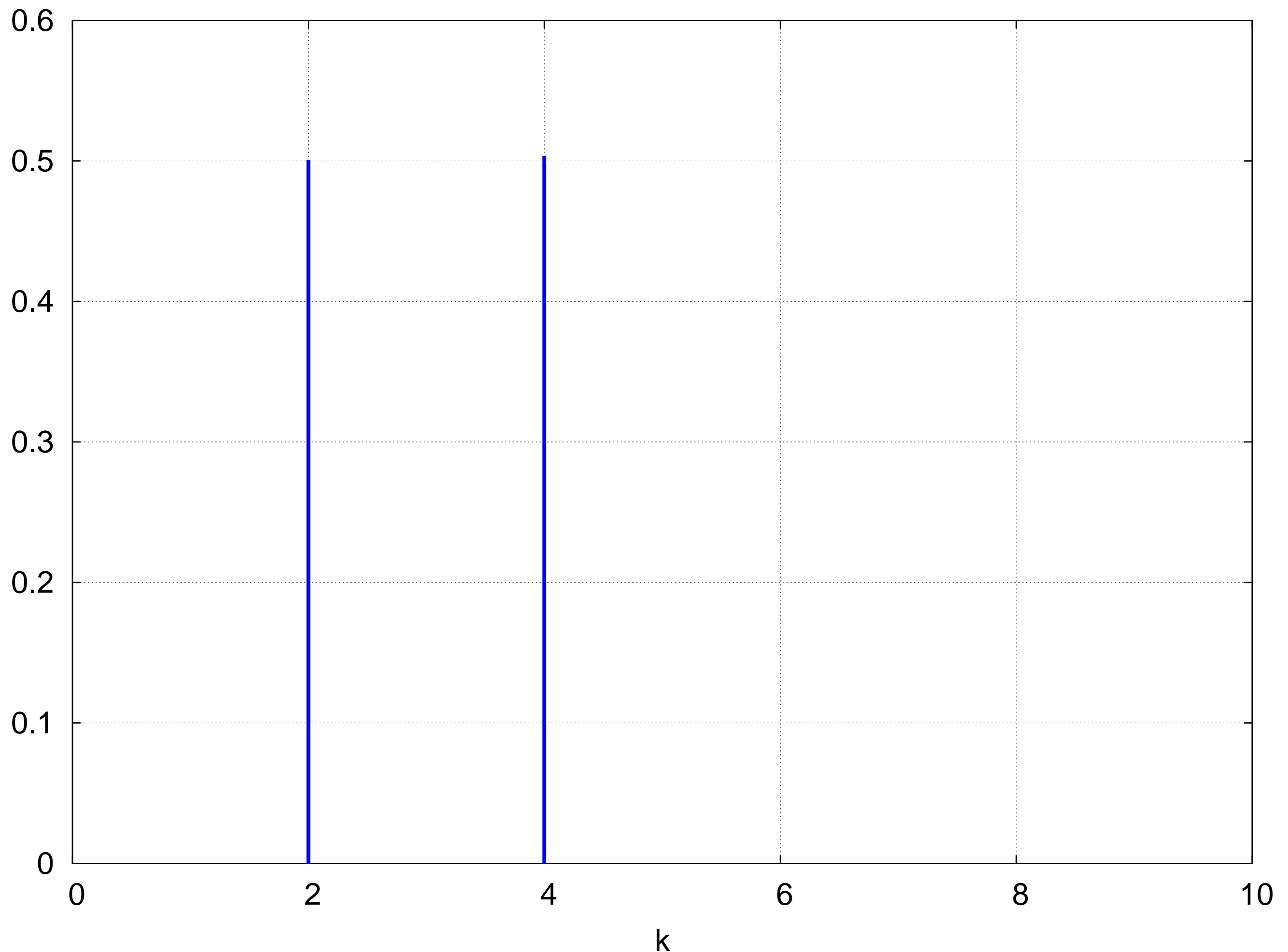

Figure 19: Chopped Frequency Spectrum for k in (0,10)

We now create a cleaned up signal list using **inverse_fft** on the chopped glist, transforming back to the time domain.

```
(%i19) flist_clean : inverse_fft ( glist_noise_chop )$
(%i20) %,fll;
(%o20)      [2.22045E-16 %i + 1.0016, 0.952 - 3.16992E-15 %i, 512]
(%i21) flist_clean : realpart(flist_clean)$
(%i22) %,fll;
(%o22)                       [1.0016, 0.952, 512]
```

Since the inverse Fourier transform will often include small imaginary parts due to floating point error, we took care to take the real part of the returned list before looking at the cleaned up signal. We now construct the list of points **[t, F_clean]**:

```
(%i23) tflist_clean : vf ( flist_clean, dt )$
(%i24) %,fll;
(%o24)                 [[0, 1.0016], [1.9961, 0.952], 512]
```

and plot first just the cleaned up signal points

```
(%i25) plot2d ([discrete,tflist_clean],[y,-2,2],
                [style,[lines,1]],[ylabel," "])$
```

which produces

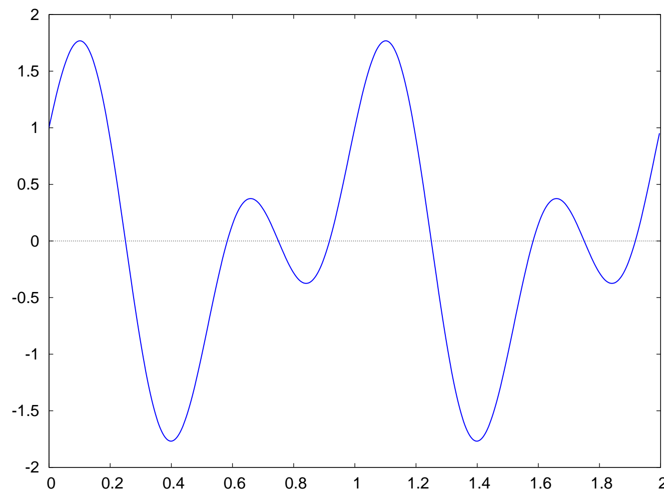

Figure 20: Cleaned Signal Points

We now show both the cleaned list points together with the original clean two frequency signal we started with, to show that the cleaned up points lie right on top of the original two frequency signal.

```
(%i26) plot2d([e ,[discrete,tflist_clean]], [t,0,2],
            [style,[lines,1],[points,1,0,1]],
             [legend,false])$
```

which produces the plot

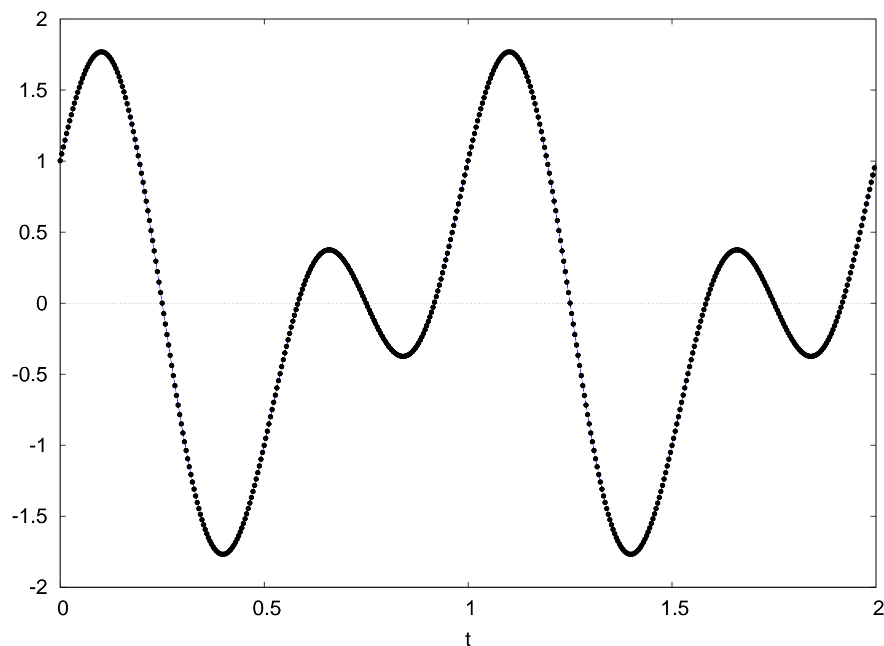

Figure 21: Cleaned Signal Points on Top of Original Clean Signal

We see that the inverse fast Fourier transform of the chopped glist frequency spectrum yields a cleaned up signal, as desired.

11.2 Our Notation for the Discrete Fourier Transform and its Inverse

Given a real valued signal $\mathbf{F(t)}$ which is to be sampled $\mathbf{N}$ times at the moments (separated by equal time intervals $\delta\mathbf{t}$) $\mathbf{t = 0}$, $\delta\mathbf{t}$, $\mathbf{2}\,\delta\mathbf{t}$, ..., $(\mathbf{N - 1})\,\delta\mathbf{t}$, one needs to select the two parameters $\mathbf{N = 2^m}$ and $\mathbf{f_s}$, where the latter is called the sampling frequency. The sampling time interval is then given by

$$\delta\mathbf{t} = \frac{\mathbf{1}}{\mathbf{f_s}} \tag{11.2}$$

(or else use the data determined value of $\delta\mathbf{t}$ to calculate $\mathbf{f_s}$). Note that the fast fourier transform algorithm used by Maxima assumes that the number of signal samples $\mathbf{N}$ is some integral power of 2, $\mathbf{N = 4, 8, 16, 32, 64}$, ... so an experimental sample might have to be padded with zeroes to achieve this condition on $\mathbf{N}$. The sampling frequency $\mathbf{f_s}$ should be greater than twice the highest frequency component to be identified in the signal and the frequency resolution $\delta\mathbf{f}$ should be smaller than the lowest frequency to be identified in the signal. Given the sampling frequency $\mathbf{f_s}$ and the number of signal samples $\mathbf{N}$, the frequency resolution $\delta\mathbf{f}$ is given by

$$\delta\mathbf{f} = \frac{\mathbf{f_s}}{\mathbf{N}} \tag{11.3}$$

We will motivate this definition below. Assuming this definition, we then require that

$$\frac{\mathbf{f_s}}{\mathbf{N}} < \mathbf{f_{low}} \tag{11.4}$$

The sampling frequency $\mathbf{f_s}$ thus needs to satisfy the two conditions:

$$\mathbf{2\,f_{high} < f_s < N\,f_{low}} \tag{11.5}$$

A convenient choice which automatically satisfies the low frequency part of these conditions is to arrange that

$$\mathbf{f_{low}} = \mathbf{n}\,\delta\mathbf{f}, \tag{11.6}$$

where $\mathbf{n = 3}$ or $\mathbf{4}$, say. Then that choice determines the frequency resolution to be used $\delta\mathbf{f} = \mathbf{f_{low}/n}$, and from the definition of $\delta\mathbf{f}$, Eq.(11.3), this requires that

$$\mathbf{f_s = N\,f_{low}/n} \tag{11.7}$$

and then Eq.(11.5) implies the condition on $\mathbf{N}$:

$$\mathbf{N} > \frac{\mathbf{2\,n\,f_{high}}}{\mathbf{f_{low}}} \tag{11.8}$$

In the simple case that $\mathbf{f_{low} = f_{high} = f_0}$ Eq.(11.7) becomes

$$\mathbf{f_s} = \mathbf{N}\,\frac{\mathbf{f_0}}{\mathbf{n}}, \tag{11.9}$$

Eq.(11.5) becomes

$$\mathbf{N > 2\,n}, \tag{11.10}$$

and

$$\delta\mathbf{f} = \frac{\mathbf{f_0}}{\mathbf{n}} \tag{11.11}$$

In Example 1, the signal frequency is $\mathbf{f_0 = 3}$, and we chose $\mathbf{n = 3}$. Then, Eq.(11.9) implies that $\mathbf{f_s = N}$, Eq.(11.10) implies that $\mathbf{N > 6}$, and Eq.(11.11) implies that $\delta\mathbf{f = 1}$. Since we need $\mathbf{N = 2^m}$ as well, we chose $\mathbf{f_s = N = 8}$.

In Example 2, $\mathbf{f_{low} = 1}$ and $\mathbf{f_{high} = 2}$, and we again chose $\mathbf{n = 3}$. Then, Eq.(11.6) implies that $\delta\mathbf{f = 1/3}$, Eq.(11.7) implies that $\mathbf{f_s = N/3}$, and Eq.(11.8) implies that $\mathbf{N > 12}$. Since we need $\mathbf{N = 2^m}$ as well, we chose $\mathbf{N = 16}$ and this forces $\mathbf{f_s = 16/3}$.

The $\mathbf{N}$ real numbers $\mathbf{F(0)}$, $\mathbf{F(\delta t)}$, $\mathbf{F(2\,\delta t)}$, ..., $\mathbf{F(m\,\delta t)}$, ..., $\mathbf{F((N-1)\,\delta t)}$ can be used to define $\mathbf{N}$ complex numbers $\mathbf{G(k\,\delta f)}$, where $\mathbf{k = 0,\ 1}$, ..., $\mathbf{(N-1)}$, according to (this incorporates Maxima's conventions):

$$\mathbf{G(k\,\delta f) = \frac{1}{N}\sum_{m=0}^{N-1} F(m\,\delta t)\, e^{-2\,\pi\, i\, m\, k/N}} \tag{11.12}$$

The Maxima conventions include where to put the factor of $\mathbf{1/N}$ and what sign to use in the exponent argument. Our notation uses $\mathbf{i}$ to stand for the pure imaginary number $\sqrt{-\mathbf{1}}$. Common engineering notation uses $\mathbf{j}$ for $\sqrt{-\mathbf{1}}$.

Equation (11.12) can be exactly inverted to arrive at an expression for the values of the original signal at $\mathbf{N}$ discrete times in terms of the $\mathbf{N}$ values of the discrete Fourier transform.

$$\mathbf{F(m\,\delta t) = \sum_{k=0}^{N-1} G(k\,\delta f)\, e^{2\,\pi\, i\, k\, m/N}} \tag{11.13}$$

where $\mathbf{m = 0,\ 1}$, ..., $\mathbf{(N-1)}$.

Using Eqs. (11.3) and (11.2), we make the replacement

$$\mathbf{\frac{1}{N} = \delta f\,\delta t} \tag{11.14}$$

in Equations (11.12) and (11.13) to get

$$\mathbf{G(f_k) = \frac{1}{N}\sum_{m=0}^{N-1} F(t_m)\, e^{-2\,\pi\, i\, f_k\, t_m}} \tag{11.15}$$

and

$$\mathbf{F(t_m) = \sum_{k=0}^{N-1} G(f_k)\, e^{2\,\pi\, i\, t_m\, f_k}} \tag{11.16}$$

where $\mathbf{f_k = k\,\delta f}$ and $\mathbf{t_m = m\,\delta t}$.

A simpler looking set of transform pairs can be achieved by letting $\mathbf{F_m = F(t_m)}$ and $\mathbf{G_k = G(f_k)}$, in terms of which Equations (11.12) and (11.13) become

$$\mathbf{G_k = \frac{1}{N}\sum_{m=0}^{N-1} F_m\, e^{-2\,\pi\, i\, m\, k/N}} \tag{11.17}$$

and

$$\mathbf{F_m = \sum_{k=0}^{N-1} G_k\, e^{2\,\pi\, i\, k\, m/N}} \tag{11.18}$$

We can use

$$\mathbf{e^{-2\,\pi\, i\, m\, k} = (e^{-2\,\pi\, i\, k})^m = (-1)^m} = 1 \tag{11.19}$$

to show that the fast Fourier amplitudes have the periodicity $\mathbf{N}$

$$\mathbf{G_{k+N} = G_k}. \tag{11.20}$$

We can now formally admit negative frequencies by letting $\mathbf{k}$ take on negative integral values, and setting $\mathbf{k = -N/2}$ we then get

$$\mathbf{G_{N/2} = G_{-N/2}}. \tag{11.21}$$

This means that the amplitude corresponding to the "Nyquist frequency"

$$\mathbf{f_{Nyquist}} = \frac{\mathbf{N}}{\mathbf{2}}\,\boldsymbol{\delta}\mathbf{f} \tag{11.22}$$

is the same complex number as the amplitude corresponding to the frequency $-\mathbf{f_{Nyquist}}$.
In the **qfft** package, the function **nyquist** calculates what we call the "Nyquist integer" $\mathbf{k_{Nyquist}}$, which is just

$$\mathbf{k_{Nyquist}} = \frac{\mathbf{N}}{\mathbf{2}}, \tag{11.23}$$

in terms of which

$$\mathbf{f_{Nyquist}} = \mathbf{k_{Nyquist}}\,\boldsymbol{\delta}\mathbf{f} \tag{11.24}$$

Likewise we can show that

$$\mathbf{G_{N/2+1}} = \mathbf{G_{-N/2+1}}. \tag{11.25}$$

which means that the amplitude corresponding to the frequency $(\mathbf{N/2+1})\,\boldsymbol{\delta}\mathbf{f} = \mathbf{f_{Nyquist}} + \boldsymbol{\delta}\mathbf{f}$ is the same complex number as the amplitude corresponding to the frequency $-\mathbf{f_{Nyquist}} + \boldsymbol{\delta}\mathbf{f}$. The only useful part of the spectrum in that contained in the frequency interval between zero and $\mathbf{f_{Nyquist}} - \boldsymbol{\delta}\mathbf{f}$, ie., in the range
$\mathbf{k} = \mathbf{0}, \mathbf{1}, \mathbf{2}, ..., (\mathbf{N/2}) - \mathbf{1}$.

In a similar manner we can show that $\mathbf{F_{m+N}} = \mathbf{F_m}$, or that $\mathbf{F}(\mathbf{t_m} + \mathbf{T}) = \mathbf{F}(\mathbf{t_m})$, where

$$\mathbf{T} = \mathbf{N}\,\boldsymbol{\delta}\mathbf{t} = \frac{\mathbf{N}}{\mathbf{f_s}} = \frac{\mathbf{1}}{\boldsymbol{\delta}\mathbf{f}}, \tag{11.26}$$

so that the fast Fourier amplitudes describe a signal which has the basic inevitable long period $\mathbf{T}$ no matter what other shorter periods (and correspondingly higher frequencies) are also present in the signal. This low frequency, long period property is an artifact of the approximate nature of Equations (11.12) and (11.13).

The fast fourier transform and its inverse should be considered as a distinct type of transform pair rather than as an approximation to either a Fourier series expansion or a Fourier integral expression of a continuous spectrum. The basic idea of the fast Fourier transform is that one has waited long enough for a physical system to "settle down" and the system is then sampled for a certain finite length of time $\mathbf{T_s}$ (we use frequency-time language only for simplicity here, the same ideas apply to wavelength-spatial domain problems).

11.3 Syntax of qfft.mac Functions

```
   FAST FOURIER TRANSFORM UTILITIES
    nyquist (ns,fs)
    sample (expr,var,ns,dvar)
    vf (flist,dvar)
    kg (glist)
    fchop (expr)
    fchop1 (expr,small)
    current_small ()
    setsmall (val)
    spectrum ( glist, nlw, ymax, k1,k2 )


1.  nyquist (ns, fs), given ns, the number of signal
    samples, and fs, the sampling frequency,
    returns the list [dt,knyq,fnyq,df] where dt is the time
    interval between function samples (dt = 1/fs), knyq in the Nyquist
    integer (knyq = ns/2),  fnyq is the Nyquist frequency
    (fnyq = fs/2 = knyq*df), and df is the frequency
    resolution (df = fs/ns) for the output of fft.
    For given ns, the values of df and dt are linked
    by the equation df*dt = 1/ns.
        Thus  nyquist(8,8); returns the list
             [0.125, 4, 4.0, 1.0], and also prints out:

        sampling interval dt =  0.125
        Nyquist integer knyq =  4
        Nyquist freq fnyq =  4.0
        freq resolution df =  1.0

2. sample (expr, var, ns, dvar) constructs a list of ns
   floating point samples F(m*dvar),
   [F(0), F(dvar), F(2*dvar),...,F((ns-1)*dvar) ],
   given the expression expr depending on var.

      sample(cos(t),t,16,1) returns a list of 16 values
      of cos(t) at intervals dt = 1, with list element
      number 1 holding cos(0), element number 2 holding
      cos(dt) = cos(1), etc.

          A signal sample list to be used with fft(flist) should
          have a length ns which is 2 raised to some integer power,
          2^3 = 8, 2^4 = 16,... If your experimental signal sample
          size does not have such a power of 2 length, you should pad
          the sample list with extra zeros.

3. vf ( flist, dvar ), given a list of function samples flist,
    consisting of ns values, and the step size dvar, returns a
        list of the form (if dvar = dt, say),
   [ [0, F(0)],[dt, F(dt)],...,[(ns-1)*dt, F( (ns-1)*dt)]]
   useful for a plot.
    For example, if the length of flist is 8, vf (flist, 1) returns the list
       [ [0, F(0)], [1, F(1)], ... , [7, F(7)] ]
```

```
4.  kg ( glist ) constructs a list of [ k, abs( g(k*df) ) ]
     for k = 0, 1, ..., knyq, which can be used with plot2d.
         knyq is the Nyquist integer, knyq = ns/2,
     where ns is the number of function samples and also the
     length of flist and glist. fchop(abs(glist[j])) is
     used to be able to plot real numbers and set tiny
         numbers to zero.

     glist is the fast Fourier transform list of complex amplitudes
         produced by glist : fft( flist ).

         Using inverse_fft( glist ) should produce flist again to within
         floating point errors. Since floating point errors will also
         introduce tiny imaginary numbers in inverse_fft ( fft (flist) )
         (if flist is real), you can use realpart(...) to recover
         a list of real numbers.

5.  fchop(expr) or fchop(list) sets tiny floating point
          numbers (with package default, less than 10^(-13) ) to zero.

6.  fchop1(expr,small) is used to override the default value of
       the small chop value with your desired value.

      Example: fchop1(s1, 1.0e-3) to set numbers smaller than
       10^(-3) to zero in the expression or list s1.

7.  current_small() returns the current default small chop value.

8.  setsmall(val) allows you to set a new value for the
     current default small chop value; use floating point
     numbers like 2.0e-3 or 2.0E-3.

9.  spectrum (glist, nlw, ymax ) creates a histogram of the frequency
    spectrum implied by glist = fft ( flist ), with line width nlw
        and vertical canvas height ymax. The range of integers
        k is 0 <= k <= knyq. The frequency associated with a given
        line is given by f = k*df, where df is the frequency
        resolution df = fs/ns. ns is the total number of signal samples
        and fs is the sampling frequency: fs = 1/dt (dt is the time
        interval between signal samples).

        spectrum (glist, nlw, ymax, k1, k2) restricts the plot to
        the range k1 <= k <= k2.
```

11.4 The Discrete Fourier Transform Derived via a Numerical Integral Approximation

Review of Trapezoidal Rule

If we let $\mathbf{f_0 = f(a),\ f_1 = f(a+h),\ f_2 = f(a+2\,h)}$, ... and $\mathbf{f_N = f(b) = f(a+N\,h)}$, then the trapezoidal rule approximation is

$$\int_{\mathbf{a}}^{\mathbf{b}} \mathbf{f(x)\,dx \approx \frac{h}{2}\,(f_0 + 2\,f_1 + 2\,f_2 + \cdots + 2\,f_{N-1} + f_N)} \tag{11.27}$$

where $\mathbf{b = a + N\,h}$ defines $\mathbf{h = (b-a)/N}$. If we now specialize to functions such that $\mathbf{f(a) = f(b)}$ then the trapezoidal rule reduces to

$$\int_{\mathbf{a}}^{\mathbf{b}} \mathbf{f(x)\,dx \approx h\ (f(a) + f(a+h) + f(a+2\,h) + \cdots + f(b-2\,h) + f(b-h))} \tag{11.28}$$

with $\mathbf{h = (b-a)/N}$.

If we now make the replacements $\mathbf{x \to t,\ a \to 0,\ b \to T,\ h = (b-a)/N \to T/N = \Delta t}$,
$\mathbf{(b-h) \to T - (T/N) = (N-1)\,\Delta t}$, then

$$\int_{\mathbf{0}}^{\mathbf{T}} \mathbf{f(t)\,dt \approx (T/N)\ (f(0) + f(\Delta t) + f(2\,\Delta t) + \cdots + f(\,(N-1)\,\Delta t)\,)} \tag{11.29}$$

or

$$\int_{\mathbf{0}}^{\mathbf{T}} \mathbf{f(t)\,dt \approx \Delta t \sum_{m=0}^{N-1} f(m\,\Delta t)} \tag{11.30}$$

where $\mathbf{\Delta t = T/N}$ and assuming $\mathbf{f(0) = f(T)}$.

A Path to the Discrete Fourier Transform

If we knew the value of a signal $\mathbf{F(t)}$ at all moments of the interval $\mathbf{0 \le t \le T}$ then we could evaluate the Fourier coefficients

$$\mathbf{C_k = \frac{1}{T}\int_0^T F(t)\,e^{-2\,\pi\,i\,k\,t/T}\,dt} \tag{11.31}$$

($\mathbf{k}$ is an integer) in terms of which the signal could be represented as the sum

$$\mathbf{F(t) = \sum_{k=-\infty}^{\infty} C_k\,e^{2\,\pi\,i\,k\,t/T}} \tag{11.32}$$

which would be a complex form of Fourier series expansion in terms of an infinite number of Fourier coefficients $\mathbf{C_k}$. We have adopted here sign and prefactor conventions which will lead us to Maxima's fast Fourier conventions.

Now assume we only know the signal at $\mathbf{N}$ discrete values $\mathbf{F(m\,\Delta t)}$, where $\mathbf{m = 0, 1, 2}, ..., \mathbf{N-1}$, and that $\mathbf{F(t) = f(t+T)}$. We can then approximate the integral in Eq.(11.31) using the trapezoidal approximation expressed by Eq.(11.30).

$$\mathbf{C_k \approx \frac{1}{T}\,(T/N) \sum_{m=0}^{N-1} F(m\,\Delta t)\,e^{-2\,\pi\,i\,k\,m\,\Delta t/T}} \tag{11.33}$$

Defining a "frequency resolution" $\boldsymbol{\delta}\mathbf{f}$ given by

$$\mathbf{\delta f = \frac{1}{T} = \frac{1}{\Delta t\,N} = \frac{f_s}{N}} \tag{11.34}$$

in which $\mathbf{f_s = 1/\Delta t}$ is the "sampling frequency", and making the replacements $\mathbf{T \to 1/\delta f,\ C_k \to G(k\,\delta f)}$

$$\mathbf{G(k\,\delta f) = \frac{1}{N} \sum_{m=0}^{N-1} F(m\,\Delta t)\,e^{-2\,\pi\,i\,(k\,\delta f)\,(m\,\Delta t)}} \tag{11.35}$$

With $\mathbf{f_k = k\,\delta f}$ and $\mathbf{t_m = m\,\Delta t}$, we can write this as

$$\mathbf{G(f_k) = \frac{1}{N}\sum_{m=0}^{N-1} F(t_m)\, e^{-2\,\pi\, i\, f_k\, t_m}} \tag{11.36}$$

which we recognise as the same as Eq.(11.15). With only $\mathbf{N}$ values of $\mathbf{F(t_m)}$ we can only determine $\mathbf{N}$ values of $\mathbf{G(f_k)}$, which by convention we take to be for the values of the integer $\mathbf{k = 0, 1, 2, ..., N-1}$.

We see from Eq.(11.34) that

$$\mathbf{\delta f\ \Delta t = \frac{1}{N}} \tag{11.37}$$

which allows Eq.(11.35) to be written as

$$\mathbf{G(k\,\delta f) = \frac{1}{N}\sum_{m=0}^{N-1} F(m\,\Delta t)\, e^{-2\,\pi\, i\, k\, m/N}} \tag{11.38}$$

which reproduces our starting point Eq.(11.12) in Sec. 11.2.

The Inverse Discrete Fourier Transform

Although the discrete Fourier transform, Eq.(11.38), is an approximation which gets better as $\mathbf{N}$ increases for fixed $\mathbf{T}$ (or since Eq.(11.34) indicates that the latter condition is equivalent to the ratio $\mathbf{f_s/N}$ being fixed, gets better as $\mathbf{N}$ and $\mathbf{f_s}$ are each increased in the same ratio), the inversion formula involves no further approximations.

Multiplying both sides of Eq.(11.38) by $\mathbf{exp(2\,\pi\, i\, k\, n/N)}$ and then summing both sides over $\mathbf{k}$, and then interchanging the order of the $\mathbf{k}$ and $\mathbf{m}$ summations on the right hand side, results in

$$\mathbf{\sum_{k=0}^{N-1} G(k\,\delta f)\, e^{2\,\pi\, i\, k\, n/N} = \frac{1}{N}\sum_{m=0}^{N-1} F(m\,\Delta t)\sum_{k=0}^{N-1} e^{2\,\pi\, i\, k\,(n-m)/N} = \frac{1}{N}\sum_{m=0}^{N-1} F(m\,\Delta t)\,(N\,\delta_{m,n}) = F(n\,\Delta t)} \tag{11.39}$$

which reproduces our discrete inverse Fourier transform formula Eq.(11.13), since $\mathbf{n}$ is an arbitrary integer.

That the sum over $\mathbf{k}$ is equal to zero if $\mathbf{m \neq n}$ can be seen by letting $\mathbf{l = n - m}$ and using $\mathbf{e^{k\,a} = (e^a)^k}$ and recognising that we have a simple geometric sum. We can also let Maxima confirm this as follows:

```
(%i1) declare([k,l,N],integer)$
(%i2) sum (exp(2*%pi*%i*k*l/N),k,0,N-1),simpsum;
                                2 %i %pi l
                              %e           - 1
(%o2)                         -----------------
                                2 %i %pi l
                                ----------
                                    N
                              %e           - 1
(%i3) %,demoivre;
(%o3)                                 0
```

In the last step the exponentials of complex arguments are converted into their trig function equivalents using

$$\mathbf{e^{i\,\theta} = \cos\,\theta + i\sin\,\theta} \tag{11.40}$$

11.5 Fast Fourier Transform References

We have consulted the treatment of the fast Fourier transform in the following books:

1. **Mathematica for the Sciences, Richard E. Crandall, Addison-Wesley, 1991**
2. **A First Course in Computational Physics, Paul L. Davies, John Wiley, 1994**
3. **Applied Mathematica: Getting Started, Getting it Done, William T. Shaw and Jason Tigg, Addison-Wesley, 1994**

Maxima by Example:
Ch. 12, Dirac Algebra and Quantum Electrodynamics *

Edwin L. Woollett

September 13, 2010

Contents

12.1 References . . . 3
12.2 High Energy Physics Notation Conventions . . . 3
12.3 Introduction to the Dirac Package . . . 6
12.4 traceConEx.mac: Examples of Frequently Used Functions in the Dirac Package . . . 7
12.4.1 Dirac Spinors . . . 12
12.5 moller0.mac: Scattering of Identical Scalar Charged Particles . . . 16
12.6 moller1.mac: High Energy Elastic Scattering of Two Electrons . . . 20
12.7 moller2.mac: Arbitrary Energy Moller Scattering . . . 26
12.8 bhabha1.mac: High Energy Limit of Bhabha Scattering . . . 30
12.9 bhabha2.mac: Arbitrary Energy Bhabha Scattering . . . 33
12.10 photon1.mac: Photon Transverse Polarization 3-Vector Sums . . . 37
12.11 Covariant Physical External Photon Polarization 4-Vectors - A Review . . . 39
12.12 compton0.mac: Compton Scattering by a Spin 0 Structureless Charged Particle . . . 39
12.13 compton1.mac: Lepton-photon scattering . . . 43
12.14 pair1.mac: Unpolarized Two Photon Pair Annihilation . . . 48
12.15 pair2.mac: Polarized Two Photon Pair Annihilation Amplitudes . . . 50
12.16 moller3.mac: Squared Polarized Amplitudes Using Symbolic or Explicit Matrix Trace Methods . . . 61
12.17 List of Dirac Package Files and Example Batch Files . . . 68

*This version uses **Maxima 5.21.1** Check **http://www.csulb.edu/~woollett/** for the latest version of these notes. Send comments and suggestions to **woollett@charter.net**

Preface

```
COPYING AND DISTRIBUTION POLICY
This document is part of a series of notes titled
"Maxima by Example" and is made available
via the author's webpage http://www.csulb.edu/~woollett/
to aid new users of the Maxima computer algebra system.

NON-PROFIT PRINTING AND DISTRIBUTION IS PERMITTED.

You may make copies of this document and distribute them
to others as long as you charge no more than the costs of printing.

These notes (with some modifications) will be published in book form
eventually via Lulu.com in an arrangement which will continue
to allow unlimited free download of the pdf files as well as the option
of ordering a low cost paperbound version of these notes.
```

Feedback from readers is the best way for this series of notes to become more helpful to new users of Maxima. *All* comments and suggestions for improvements will be appreciated and carefully considered.

```
LOADING FILES
The defaults allow you to use the brief version load(fft) to load in the
Maxima file fft.lisp.
To load in your own file, such as qxxx.mac
using the brief version load(qxxx), you either need to place
qxxx.mac in one of the folders Maxima searches by default, or
else put a line like:

file_search_maxima : append(["c:/work2/###.{mac,mc}"],file_search_maxima )$

in your personal startup file maxima-init.mac (see later in this chapter
for more information about this).

Otherwise you need to provide a complete path in double quotes,
as in load("c:/work2/qxxx.mac"),

We always use the brief load version in our examples, which are generated
using the XMaxima graphics interface on a Windows XP computer, and copied
into a fancy verbatim environment in a latex file which uses the fancyvrb
and color packages.
```

```
Maxima.sourceforge.net. Maxima, a Computer Algebra System. Version 5.18.1
 (2009). http://maxima.sourceforge.net/
```

The homemade function **fll(x)** (first, last, length) is used to return the first and last elements of lists (as well as the length), and is automatically loaded in with **mbe1util.mac** from Ch. 1. We will include a reference to this definition when working with lists.

This function has the definitions

```
fll(x) := [first(x),last(x),length(x)]$
declare(fll,evfun)$
```

Some of the examples used in these notes are from the Maxima html help manual or the Maxima mailing list:
http://maxima.sourceforge.net/maximalist.html.

The author would like to thank the Maxima developers for their friendly help via the Maxima mailing list.

12.1 References

The following works are useful sources. Some abbreviations (which will be used in the following sections) are defined here.

- Aitchison, I.J.R., Relativistic Quantum Mechanics, Barnes and Noble,1972
- A/H: Aitchison,I.J.R, and Hey, A.J.G., Gauge Theories in Particle Physics, Adam Hilger, 1989
- B/D: Bjorken, James D. and Drell, Sidney D, Relativistic Quantum Mechanics, McGraw Hill, 1964
- BLP: Berestetskii, V. B, Lifshitz, E. M., and Pitaevskii, L. P., Quantum Electrodynamics, Course of Theoretical Physics, Vol. 4, 2nd. Ed. Pergamon Press, 1982
- Berestetskii, V. B, Lifshitz, E. M., and Pitaevskii, L. P., Relativistic Quantum Theory, Part 1, Course of Theoretical Physics, Vol. 4, Pergamon Press, 1971
- De Wit, B. and Smith, J., Field Theory in Particle Physics, North-Holland, 1986
- G/R: Greiner, Walter, and Reinhardt, Joachim, Quantum Electrodynamics, fourth ed., Springer, 2009
- Griffiths, Introduction to Elementary Particles, Harper and Row, 1987
- H/M: Halzen, Francis and Martin, Alan D., Quarks and Leptons, John Wiley, 1984
- I/Z: Itzykson, Claude and Zuber, Jean-Bernard, Quantum Field Theory, McGraw-Hill, 1980
- Jauch, J.M., and Rohrlich, F., The Theory of Photons and Electrons, Second Expanded Edition, Springer-Verlag, 1976
- Kaku, Michio, Quantum Field Theory, Oxford Univ. Press, 1993
- Maggiore, Michele, A Modern Introduction to Quantum Field Theory, Oxford Univ. Press, 2005
- P/S: Peskin, Michael E. and Schroeder, Daniel V., An Introduction to Quantum Field Theory, Addison-Wesley, 1995
- Quigg, Chris, Gauge Theories of the Strong, Weak, and Electromagnetic Interactions, Benjamin/Cummings, 1983
- Renton, Peter, Electroweak Interactions, Cambridge Univ. Press, 1990
- Schwinger, Julian, Particles, Sources, and Fields, Vol. 1, Addison-Wesley, 1970
- Sterman, George, An Introduction to Quantum Field Theory, Cambridge Univ. Press, 1993
- Weinberg, Steven, The Quantum Theory of Fields, Vol.I, Cambridge Univ. Press, 1995

12.2 High Energy Physics Notation Conventions

We use the same conventions as P/S (Peskin and Schroeder) and Maggiore. The electromagnetic units are Heaviside-Lorentz, and natural units are used, so $\mathbf{e^2} = \boldsymbol{4\pi\alpha}$.

The only exception is that we interpret the Feynman rules as an expression for the quantity $-i\,M$, following the conventions of Griffiths, and Halzen/Martin.

The diagonal elements of the metric tensor are $(1, -1, -1, -1)$, and the helicity (chiral) representation used for explicit matrix methods with the Dirac gamma matrices is (in 2 $\times$ 2 block form)

$$\gamma^0 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \gamma^j = \begin{pmatrix} 0 & \sigma^j \\ -\sigma^j & 0 \end{pmatrix}, \quad \gamma^5 = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}, \tag{12.1}$$

in which σ^j for $j = 1, 2, 3$ are the 2×2 Pauli matrices

$$\sigma^1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \sigma^2 = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad \sigma^3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{12.2}$$

The above form for γ^5 is equivalent to

$$\gamma^5 = i\, \gamma^0\, \gamma^1\, \gamma^2\, \gamma^3. \tag{12.3}$$

The lepton Dirac spinor is the four element column matrix $u(p, \sigma)$ where $\sigma = +1$ picks out positive helicity $h = +1/2$ and $\sigma = -1$ picks out negative helicity $h = -1/2$. The corresponding barred particle spinor is a four element row matrix

$$\bar{u}(p, \sigma) = u^\dagger\, \gamma^0 \tag{12.4}$$

and the particle Dirac spinor normalization is

$$\bar{u}(p, \sigma)\, u(p, \sigma') = 2\, m\, \delta_{\sigma\, \sigma'}, \quad \sigma,\, \sigma' = \pm 1, \tag{12.5}$$

and we also have

$$\sum_\sigma u(p, \sigma)\, \bar{u}(p, \sigma) = \not{p} + m, \tag{12.6}$$

in which a 4×4 unit matrix is understood to be multiplying the scalar mass symbol m, and $\not{p} = p_\mu\, \gamma^\mu$ with use of the summation convention.

The antilepton Dirac spinor is the four element column matrix $v(p, \sigma)$ in which $\sigma = +1$ picks out the positive helicity $h = +1/2$ antiparticle state and $\sigma = -1$ picks out negative helicity $h = -1/2$ antiparticle state, is defined (with P/S phase choice) via

$$v(p, \sigma) = -\gamma^5\, u(p, -\sigma) \tag{12.7}$$

The corresponding barred antiparticle spinor is a four element row matrix

$$\bar{v}(p, \sigma) = v^\dagger\, \gamma^0 \tag{12.8}$$

and the normalization is

$$\bar{v}(p, \sigma)\, v(p, \sigma') = -2\, m\, \delta_{\sigma\, \sigma'} \quad \sigma,\, \sigma' = \pm 1. \tag{12.9}$$

We also have

$$\sum_\sigma v(p, \sigma)\, \bar{v}(p, \sigma) = \not{p} - m \tag{12.10}$$

We define the 4×4 the spin projection matrix

$$S(p, \sigma) = \frac{1}{2}\, (1 + \sigma\, \gamma^5\, \not{s}_R(p)) \tag{12.11}$$

in which the 1 is interpreted as the 4×4 unit matrix and in which $\not{s}_R(p) = s_{R\mu}\, \gamma^\mu$, and the right handed spin 4-vector s_R^μ generated by the 4-vector p has the components

$$s_r^\mu(p) = \left(\frac{|\mathbf{p}|}{m},\, \frac{E}{m}\, \hat{\mathbf{p}} \right) \tag{12.12}$$

in which $E = \sqrt{m^2 + \mathbf{p}^2}$.

For finite (non-zero) mass m we have

$$u(p,\sigma)\,\bar{u}(p,\sigma) = S(p,\sigma)\,(\not{p} + m) = (\not{p} + m)\,S(p,\sigma) \tag{12.13}$$

and again for finite mass,

$$v(p,\sigma)\,\bar{v}(p,\sigma) = S(p,\sigma)\,(\not{p} - m) = (\not{p} - m)\,S(p,\sigma) \tag{12.14}$$

In the case of zero mass leptons

$$u(p,\sigma)\,\bar{u}(p,\sigma) = \frac{1}{2}\,(1 + \sigma\,\gamma^5)\,\not{p} \tag{12.15}$$

and for zero mass antiparticles

$$v(p,\sigma)\,\bar{v}(p,\sigma) = \frac{1}{2}\,(1 - \sigma\,\gamma^5)\,\not{p}. \tag{12.16}$$

Calculation of unpolarized cross sections requires summing the absolute value squared of a polarized amplitude over the helicity quantum numbers of the particles involved, and the result can be written as the matrix trace of a 4×4 matrix. A simple example follows. We use a compressed notation in which, for example, u_1 stands for $u(p_1, \sigma_1)$ and u_2 stands for $u(p_2, \sigma_2)$. Let Γ be an arbitrary 4×4 matrix.

One can show that

$$(\bar{u}_2\,\Gamma\,u_1)^* = \bar{u}_1\,\bar{\Gamma}\,u_2 \tag{12.17}$$

in which

$$\bar{\Gamma} = \gamma^0\,\Gamma^\dagger\,\gamma^0. \tag{12.18}$$

Then

$$\sum_{\sigma_1\,\sigma_2} |\bar{u}_2\,\Gamma\,u_1|^2 = \sum_{\sigma_1\,\sigma_2} \bar{u}_2\,\Gamma\,u_1\,\bar{u}_1\,\bar{\Gamma}\,u_2 \tag{12.19}$$

We use the sum over σ_1 to replace $u_1\,\bar{u}_1$ by $(\not{p}_1 + m)$, and are left with

$$\sum_{\sigma_2} \bar{u}_2\,\hat{\Gamma}\,u_2, \quad \hat{\Gamma} \doteq \Gamma\,(\not{p}_1 + m)\,\bar{\Gamma}. \tag{12.20}$$

Writing out the matrix indices explicitly, the sum becomes (using the summation convention for repeated matrix indices)

$$\sum_{\sigma_2} \bar{u}_{2\,a}\,\hat{\Gamma}_{a\,b}\,u_{2\,b} = \hat{\Gamma}_{a\,b} \sum_{\sigma_2} u_{2\,b}\,\bar{u}_{2\,a} = \hat{\Gamma}_{a\,b}\,(\not{p}_2 + m)_{b\,a} = \textbf{Trace}\,\{\hat{\Gamma}\,(\not{p}_2 + m)\} \tag{12.21}$$

Note that **Trace** $(A\,B) =$ **Trace** $(B\,A)$. We then have

$$\sum_{\sigma_1\,\sigma_2} |\bar{u}_2\,\Gamma\,u_1|^2 = \textbf{Trace}\,\{(\not{p}_2 + m)\,\Gamma\,(\not{p}_1 + m)\,\bar{\Gamma}\}. \tag{12.22}$$

12.3 Introduction to the Dirac Package

Three independent Dirac algebra methods are available in our package. One can gain confidence in an unexpected result by doing the same calculation in multiple ways. The explicit Dirac spinor methods, which use an explicit representation of the gamma matrices, are bug free, fast, and the route to polarized amplitudes (rather that the square of polarized amplitudes). Moreover, summing the absolute square of the polarized amplitudes over all helicity values leads quickly and reliably to the unpolarized cross section in terms of the frame dependent kinematic variables such as scattering angle. However the version available with this package is not a covariant method and is always frame dependent. The explicit Dirac matrix methods using the Maxima function **mat_trace** are inherently less bug prone and also are faster in execution than the purely symbolic methods. However, the purely symbolic contraction and trace methods available in this package are capable of introducing the kinematic invariants s, t, and u into the calculation in a simple way, or to reducing the types of four vector dot products to two, leading to useful covariant results. In the following examples, we usually use the fastest and most convenient method for a given problem. An example of using the **matrix** trace and contraction methods, and the **symbolic** trace and contraction methods to generate the square of polarized amplitudes is provided in the section dealing with the batch file **moller3.mac**.

Each physics calculation is written in the form of a batch file with an appropriate name, such as **moller1.mac**. Although one can start a problem calculation in two steps by first loading the Dirac package with **load("dirac.mac")** and then, for example, launching the batch file using **batch ("moller1.mac")**, it is easier during the process of experimentation to have one file called, say, **work.mac** which contains both of these commands. With this latter system, one can redo an edited batch file by restarting Maxima, and then using **Alt+P** to recall the last command (if the last command was **load("work.mac")**), or to type in **load("work.mac")**. This can be made even easier by defining the function **w() := load ("work.mac")** in the file **work.mac**. The later versions of Maxima will not print out the batch command (when using the **work.mac** file method) but rather will print the message **read and interpret file: #pc:/work4/moller1.mac** for example.

The batch files use some private code in the author's startup file **maxima-init.mac** to print out the year-month-day of the batch file run, and also to print the Maxima version number. The startup file code fragments are

```
 load("new-timedate.lisp")$
 mydate : apply ('sconcat, rest (charlist (timedate(+7)),-15))$
_binfo% : "Maxima 5.21.1"$
```

The lisp file **new-timedat.lisp**, available on the author's webpage, was written by Maxima developer Leo Butler, and allows Windows users to overcome a bug to get a year-month-day stamp using Maxima's **timedate** function. One can then, for example, print out the year-month-day at the start of a Maxima session, using

```
  disp (mydate)$
```

as a line in the startup file.

In our batch files is a line

```
   print ("      ver: ",_binfo%,"  date: ",mydate )$
```

which prints out the Maxima version number and year-month-day as:

```
     ver:  Maxima 5.21.1   date:  2010-09-01
```

12.4 traceConEx.mac: Examples of Frequently Used Functions in the Dirac Package

Once you have loaded the Dirac package (which includes loading five files in addition to **dirac.mac**), you can find traces using **tr** (for symbolic traces), or **Gtr** (for symbolic traces using a **G** argument), or **nc_tr**, equivalent to **noncov (tr(..))** which begins with a symbolic trace and ends with a reduction to frame dependent quantities.

For the trace of explicit matrix expressions, the functions **m_tr** and **mat_trace** have a different syntax but both do the same ultimate job, since the **m_tr** form is converted automatically into the **mat_trace** form. To obtain useful expressions, one should first use **comp_def** to define the frame dependent components of relevant 4-vectors, and combine **mcon** with either **m_tr** or **mat_trace**.

You can calculate contractions using **Con** (for any type of argument), **scon** (for symbolic contraction), **mcon** (for contraction of expressions involving explicit matrices), and **econ** (for contraction of expressions produced by **noncov**).

D(p,q) = D(q,p) stands for the 4-vector inner product $p \cdot q = q \cdot p$.

Gm(mu,nu) stands for $g^{\mu\nu}$ or $g_{\mu\nu}$ depending on the context.

UI(p,mu) stands for p^{μ} and **LI(p,mu)** stands for p_{μ}. With our metric convention, $p_0 = p^0$.

comp_def (p1(a,b,c,d)) creates an array with the name **p1** whose elements can be viewed using **listarray (p1)** and whose elements can be accessed by using **p1[0]** to get the value **a**. The array element **p1[mu]** (with **mu = 0,1,2,3**) stands for p_1^{μ}. **comp_def** is used to assign components to 4-vectors in a given frame of reference.

Eps(mu,nu,rh,la) stands for $\epsilon^{\mu\nu\rho\lambda}$, but the antisymmetry property only comes into play with **noncov(Eps(a,b,c,d)) --> eps4[a,b,c,d]**, with the convention that $\epsilon^{0\,1\,2\,3} = 1$. Dummy indices **N1, N2, ...** are automatically generated when needed and are eventually summed over using **noncov**. Any contraction involving **Eps** expressions is automatically done using **noncov** which converts any **Eps(a,b,c,d)** to the array **eps4[a,b,c,d]** (which is the quantity defined to be antisymmetric), and converts **Gm(a,b)** to the array **gmet[a,b]** (which is delared as symmetric).

G(mu,p,nu,q) is algebraically equivalent to the matrix

Gam[mu] . sL(p) . Gam[nu] . sL(q)

in which the Lorentz indices **mu** and **nu** take on possible values **0,1,2,3**, and **sL(p)** is the Feynman slash matrix generated for the 4-vector labeled **p**. This interpretation depends on the content of the package list **indexL**, which tells the package functions which symbols are to be treated as Lorentz indices. Any other symbol is treated as a symbol for a four-momentum, unless that symbol appears on the list **massL** (in which case the symbol is treated as a mass), or unless the symbol has been declared a scalar with the command, for example using the package function **mscalar**: **mscalar (c1,c2,c3)$**, which not only declares **c1,..** scalars but puts them on the list **scalarL**. However, **G(mu,p,nu,q)** is treated as a purely symbolic expression in this package and is not represented as an explicit matrix. In fact, the function **Gtr** simply replaces the symbol **G** by the symbol **tr** and evaluates the result. Further mass symbols can be placed on **massL** via the package function **mass()**, as in **mass (m1,m2)**.

We illustrate some key functions at work with the batch file **traceConEx.mac**, (using a **work.mac** file to first load the dirac package and then the batch file of examples). Note that **tr** automatically contracts on repeated Lorentz indices.

```
(%i1) load(work)$
                              work4 dirac.mac
                                   dgexp
                                   dgcon
                                  dgtrace
                                  dgeval
                                  dgmatrix
massL = [m,M]
```

```
indexL = [la,mu,nu,rh,si,ta,al,be,ga]
scalarL = []
" reserved program capital letter name use: "
" Chi, Con, D, Eps, G, G5, Gam, Gm, LI, Nlist, Nlast, UI, P, S, Sig"
"  UU, VP, VV, I2, Z2, CZ2, I4, Z4, CZ4, RZ4, N1,N2,... "

read and interpret file: #pc:/work4/traceConEx.mac
(%i2) " ======================================================="
(%i3) "  file traceConEx.mac "
(%i4) "  Maxima by Example, Ch. 12 "
(%i5) "  Dirac Algebra and Quantum Electrodynamics "
(%i6) "  Edwin L Woollett, woollett@charter.net "
(%i7) "  http://www.csulb.edu/~woollett  "
(%i8) print("      ver: ",_binfo%,"  date: ",mydate)
      ver:  Maxima 5.21.1   date:  2010-09-01

(%i9) " ======================================================="
(%i10) " Examples of Symbolic Traces "
(%i11) " ---------------------------------------------------"
(%i12) " Symbolic contraction on repeated Lorentz indices is "
(%i13) " automatically carried out by the tr and Gtr functions."
(%i14) " Symbols appearing on the list indexL are recognised"
(%i15) " as Lorentz indices."
(%i16) "  You can add symbols to indexL using function index."
(%i17) " You can remove symbols from indexL using function unindex."
(%i18) " ---------------------------------------------------"
(%i19) " Three examples of using mat_trace for the trace of an "
(%i20) " explicit matrix expression are also given."
(%i21) " Gam[mu] for mu = 0,1,2,3, and Gam[5] are explicit Dirac matrices"
(%i22) " Z4 is the 4 x 4 zero matrix, I4 is the 4 x 4 identity matrix"
(%i23) " defined by the Dirac package."
(%i24) indexL
(%o24) [la,mu,nu,rh,si,ta,al,be,ga]
(%i25) tr(mu,mu)
(%o25) 16
(%i26) Gtr(G(mu,mu))
(%o26) 16
(%i27) tr(mu,nu)
(%o27) 4*Gm(mu,nu)
(%i28) noncov(%)
(%o28) 4*gmet[mu,nu]
(%i29) Gtr(G(mu,nu))
(%o29) 4*Gm(mu,nu)
(%i30) " Examples of mat_trace "
(%i31) I4
(%o31) matrix([1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1])
(%i32) mat_trace(I4)
(%o32) 4
(%i33) Z4
(%o33) matrix([0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0])
(%i34) mat_trace(Z4)
(%o34) 0
(%i35) a(mu,nu):=mat_trace(Gam[mu] . Gam[nu])
(%i36) b(mu,nu):=4*gmet[mu,nu]
(%i37) c(mu,nu):=is(equal(a(mu,nu),b(mu,nu)))
(%i38) c(0,0)
(%o38) true
(%i39) for mu from 0 thru 3 do
           (for nu from 0 thru 3 do
                if not c(mu,nu) then print(mu,nu,"  false"))
(%i40) tr(mu,p,q,mu)
(%o40) 16*D(p,q)
```

```
(%i41) Gtr(G(mu,p,q,mu))
(%o41) 16*D(p,q)
(%i42) tr(G5,mu,nu,rh,la)
(%o42) -4*%i*Eps(mu,nu,rh,la)
(%i43) noncov(%)
(%o43) 4*%i*eps4(la,mu,nu,rh)
(%i44) Gtr(G(G5,mu,nu,rh,la))
(%o44) -4*%i*Eps(mu,nu,rh,la)
(%i45) tr(G5,p,mu,nu,rh)
(%o45) -4*%i*LI(p,N1)*Eps(N1,mu,nu,rh)
(%i46) " Symbols for masses, like m, which appear on the list massL are treated"
(%i47) " differently, and automatic expansion is carried out."
(%i48) massL
(%o48) [m,M]
(%i49) tr(m+p1,mu,m+p2,mu)
(%o49) 16*m^2-8*D(p1,p2)
```

and examples of factoring out scalar constants, if declared:

```
(%i50) " ======================================================="
(%i51) " The symbolic trace will recognise scalar constants multiplying"
(%i52) " momentum symbols and factor them out correctly. By using "
(%i53) " mscalar to declare scalar constant symbols, you maintain "
(%i54) " a list scalarL, and you can remove the scalar property "
(%i55) " using unscalar "
(%i56) scalarL
(%o56) []
(%i57) mscalar(c1,c2)
scalarL = [c2,c1]

(%o57) done
(%i58) tr(c1*p,q)
(%o58) 4*c1*D(p,q)
(%i59) tr(c1*p,c2*q)
(%o59) 4*c1*c2*D(p,q)
(%i60) tr(m+c1*p,m+c2*q)
(%o60) 4*c1*c2*D(p,q)+4*m^2
(%i61) tr(G5,c1*p,mu,nu,rh)
(%o61) -4*%i*c1*LI(p,N2)*Eps(N2,mu,nu,rh)
(%i62) unscalar(c1)
scalarL = [c2]
(%i63) tr(c1*p,q)
(%o63) 4*D(c1*p,q)
```

and examples of **symbolic contractions** and **contractions of explicit matrices**:

```
(%i64) " ========================================================"
(%i65) "Examples of Symbolic and Explicit Matrix Contractions "
(%i66) " Symbolic contractions using Con or scon do not require "
(%i67) " the contraction indices, if automatic contraction over "
(%i68) " all repeated Lorentz indices is desired."
(%i69) " Contractions of explicit matrix expressions require"
(%i70) " the contraction indices be specified, as in mcon (e,mu,nu)"
(%i71) " Remember that Con is (hopefully) designed to select either"
(%i72) " scon, econ, or mcon depending on the type of expression."
(%i73) Con(G(mu,mu))
(%o73) 4
(%i74) " Con fails if you don't supply the intended contraction "
(%i75) " index is the expression is an explicit matrix expression."
(%i76) Con(Gam[mu] . Gam[mu])
(%o76) Gam[mu]^^2
```

```
(%i77) "But providing the contraction index, we get the correct result."
(%i78) Con(Gam[mu] . Gam[mu],mu)
(%o78) matrix([4,0,0,0],[0,4,0,0],[0,0,4,0],[0,0,0,4])
(%i79) is(equal(Con(Gam[mu] . Gam[mu],mu),4*I4))
(%o79) true
(%i80) Con(G(mu,nu,nu,mu))
(%o80) 16
(%i81) Con(Gam[mu] . Gam[nu] . Gam[nu] . Gam[mu],mu,nu)
(%o81) matrix([16,0,0,0],[0,16,0,0],[0,0,16,0],[0,0,0,16])
(%i82) is(equal(Con(Gam[mu] . Gam[nu] . Gam[nu] . Gam[mu],mu,nu),16*I4))
(%o82) true
(%i83) " Control symbolic contractions by providing contraction indices."
(%i84) Con(G(mu,nu,nu,mu),mu)
(%o84) 4*Gm(nu,nu)
(%i85) Con(G(mu,nu,nu,mu),nu)
(%o85) 4*G(mu,mu)
(%i86) Con(G(mu,p,q,mu))
(%o86) 4*D(p,q)
(%i87) Con(G(p,mu,mu,q))
(%o87) 4*G(p,q)
(%i88) Con(G(p,mu,mu,p))
(%o88) 4*D(p,p)
(%i89) Con(UI(p,mu)*UI(q,mu))
(%o89) D(p,q)
(%i90) Con(UI(p,mu)*Gm(mu,nu))
(%o90) UI(p,nu)
(%i91) Con(G(a,mu,b,c,mu,a))
(%o91) 4*D(a,a)*D(b,c)
(%i92) Con(G(G5,mu,nu,nu,mu))
(%o92) 16*G(G5)
(%i93) Con(Gam[5] . Gam[mu] . Gam[nu] . Gam[nu] . Gam[mu],mu,nu)
(%o93) matrix([-16,0,0,0],[0,-16,0,0],[0,0,16,0],[0,0,0,16])
(%i94) is(equal(Con(Gam[5] . Gam[mu] . Gam[nu] . Gam[nu] . Gam[mu],mu,nu),
                16*Gam[5]))
(%o94) true
```

and examples of **noncov**, **comp_def**, and **VP**:

```
(%i95) " ======================================================"
(%i96) " Examples of Use of noncov, comp_def, VP "
(%i97) noncov(D(p,q))
(%o97) -p[3]*q[3]-p[2]*q[2]-p[1]*q[1]+p[0]*q[0]
(%i98) noncov(Gm(mu,nu))
(%o98) gmet[mu,nu]
(%i99) " gmet is a two dimensional numeric array "
(%i100) gmet[0,0]
(%o100) 1
(%i101) noncov(UI(p,mu))
(%o101) p[mu]
(%i102) noncov(LI(p,mu))
(%o102) p[3]*gmet[3,mu]+p[2]*gmet[2,mu]+p[1]*gmet[1,mu]+p[0]*gmet[0,mu]
(%i103) noncov(LI(p,N1)*Eps(N1,1,2,3))
(%o103) p[0]
(%i104) noncov(Eps(mu,nu,rh,la))
(%o104) -eps4(la,mu,nu,rh)
(%i105) " eps4 is declared antisymmetric "
(%i106) eps4(mu,nu,rh,la)
(%o106) -eps4(la,mu,nu,rh)
(%i107) " eps4 is a two dimensional numeric array "
(%i108) eps4[0,1,2,3]
(%o108) 1
```

```
(%i109) "Use comp_def to specify 4-vec. components in a given frame "
(%i110) " Syntax: comp_def (pn (energy symbol,3-mom_x,3-mom_y,3-mom_z)) "
(%i111) " or, comp_def ( pa(Ea,pax,pay,paz), pb (Eb,pbx,pby,pbz ),...)"
(%i112) " Here, we assume scattering in the z-x plane."
(%i113) assume(E > 0,th >= 0,th <= %pi)
(%i114) comp_def(p1(E,0,0,E),p2(E,0,0,-E),p3(E,E*sin(th),0,E*cos(th)),
                  p4(E,-E*sin(th),0,-E*cos(th)))
(%i115) listarray(p1)
(%o115) [E,0,0,E]
(%i116) p1[0]
(%o116) E
(%i117) p1[3]
(%o117) E
(%i118) noncov(D(p1,p2))
(%o118) 2*E^2
(%i119) noncov(D(p2+p1,p2+p1))
(%o119) 4*E^2
(%i120) " can also use VP for 4-vector product in terms of components"
(%i121) VP(p2+p1,p2+p1)
(%o121) 4*E^2
(%i122) factor(VP(p1-p3,p1-p3))
(%o122) 2*(cos(th)-1)*E^2
```

and examples of **nc_tr**, **econ**, **mcon**, **m_tr**, and **mat_trace**:

```
(%i123) " ======================================================="
(%i124) " Comparing econ,nc_tr, mcon, m_tr, mat_trace Methods "
(%i125) Con_comp:Con(nc_tr(p3,mu,p1,nu)*nc_tr(p4,mu,p2,nu),mu,nu)
(%o125) 64*sin(th)^2*E^4+96*cos(th)^2*E^4+64*cos(th)*E^4+96*E^4
(%i126) Con_comp-econ(nc_tr(p3,mu,p1,nu)*nc_tr(p4,mu,p2,nu),mu,nu)
(%o126) 0
(%i127) Con_comp-Con(m_tr(p3,mu,p1,nu)*m_tr(p4,mu,p2,nu),mu,nu)
(%o127) 0
(%i128) Con_comp-mcon(m_tr(p3,mu,p1,nu)*m_tr(p4,mu,p2,nu),mu,nu)
(%o128) 0
(%i129) Con_comp-Con(mat_trace(sL(p3) . Gam[mu] . sL(p1) . Gam[nu])
                   *mat_trace(sL(p4) . Gam[mu] . sL(p2) . Gam[nu]),mu,nu)
(%o129) 0
(%i130) Con_comp-mcon(mat_trace(sL(p3) . Gam[mu] . sL(p1) . Gam[nu])
                    *mat_trace(sL(p4) . Gam[mu] . sL(p2) . Gam[nu]),mu,nu)
(%o130) 0
```

It is easy to verify two properties of **Gam[5]** using explicit matrices.

```
(%i131) " ======================================================="
(%i132) " Gam[5] has zero trace and Gam[5].Gam[5] = I4 "
(%i133) mat_trace(Gam[5])
(%o133) 0
(%i134) is(equal(Gam[5] . Gam[5],I4))
(%o134) true
```

12.4.1 Dirac Spinors

Finally, we present an example of an expression constructed from **Dirac spinors** and matrices for given helicity quantum numbers.

We generate an explicit Dirac "spinor", a four element column matrix, corresponding to $\mathbf{u}(\mathbf{p1},\sigma)$ using the function `UU(E,p,theta,phi,sv)` which descibes a spin `1/2` particle with relativistic energy E, 3-momentum magnitude p, whose 3-vector momentum is described by polar angles `(theta,phi)`, where `0 <= theta <= %pi` and `0 <= phi < 2*%pi`, and whose helicity quantum number `sv` is `1` for helicity `+1/2` and `sv` is `-1` for helicity `-1/2`.

We normally describe scattering in the `z-x` plane, and if `p1` makes an angle `th <= %pi` with the positive `z` axis and we are describing "positive" helicity (`sv = 1`) (or "righthanded, R") we would use the syntax
`up1 : UU(E1,p1,th,0,1)`.

If a different particle is moving in the `z-x` plane in a direction opposite to `p1` with 3-momentum `p2` and energy `E2`, we would use the syntax (taking `sv2 = +1`)
`up2 : UU (E2,p2,%pi - th, %pi, 1)`. For a negative helicity particle, we would use `sv = -1`.

A barred particle spinor (a row matrix) is generated by using `sbar` which has the definition
`sbar (_uu%) := hc (_uu%) . Gam[0]$`

in which `hc` is the package defined hermitian conjugate function.

We would use the syntax `up1b : sbar (UU(E1,p1,th,0,1))` for example.

To generate the antiparticle spinor corresponding to $\mathbf{v}(\mathbf{p},\sigma)$ for an antiparticle of energy `E1` and 3-momentum `p1` we use the function
`VV(E,p,theta,phi,sv)`.
We normally describe scattering in the `z-x` plane, and if `p1` makes an angle `th <= %pi` with the positive `z` axis and we are describing "positive" helicity (`sv = 1`) (or "righthanded, R") we would use the syntax
`vp1 : VV(E1,p1,th,0,1)`. For a negative helicity antiparticle, we would use `sv = -1`.

If a different antiparticle is moving in the `z-x` plane in a direction opposite to `p1` with 3-momentum `p2` and energy `E2`, we would use the syntax (taking for example `sv2 = +1`)
`vp2 : VV (E2,p2,%pi - th, %pi, 1)`.

A barred antiparticle spinor (a row matrix) is generated by using `sbar` with the syntax
`vp1b : sbar (VV (E1,p1,th,0,1))` for example.

In the following example, we construct spinors needed to calculate the matrix element for the `RL --> RL` process
`e(-,p1,1) + e(+,p2,-1) --> mu(-,p3,1) + mu(+,p4,-1)`
in which the electron and muon have positive helicity and the positron and antimuon have negative helicity. Since in the high energy limit the energy and 3-momentum magnitude are the same, we replace `p` by `E`, and we are working in the center of momentum frame, so each "particle" has the same energy.

```
( up1 : UU(E,E,0,0,1), vp2b : sbar (VV (E,E,%pi,0,-1)),
                a12 : vp2b.Gam[mu].up1,
            up3b : sbar (UU (E,E,th,0,1)), vp4 : VV (E,E,%pi-th,%pi,-1),
                a34 : up3b.Gam[mu].vp4)$
```

Ignoring the factor `-e^2`, the numerator `Mn` of the transition amplitude is given by the contraction of the product `a12*a34` on the index `mu`.

```
(%i135) " ======================================================="
(%i136) " Contraction example involving Dirac spinors"
(%i137) " High energy limit in center of momentum frame of "
(%i138) " e(-,p1,1) + e(+,p2,-1) --> mu(-,p3,1) + mu(+,p4,-1) "
(%i139) "    sv1 = 1, sv2 = -1, sv3 = 1, sv4 = -1  "
(%i140) " ref: P/S  p. 131 ff  "
(%i141) (up1:UU(E,E,0,0,1),vp2b:sbar(VV(E,E,%pi,0,-1)),
         a12:vp2b . Gam[mu] . up1,up3b:sbar(UU(E,E,th,0,1)),
         vp4:VV(E,E,%pi-th,%pi,-1),a34:up3b . Gam[mu] . vp4)
(%i142) " The numerator of the amplitude, ignoring -e^2 factor"
(%i143) Mn:Con(a12*a34,mu)
(%o143) 8*cos(th/2)^2*E^2
(%i144) " the denominator of the amplitude "
(%i145) s_th:VP(p2+p1,p2+p1)
(%o145) 4*E^2
(%i146) Amp_RL_RL:Mn/s_th
(%o146) 2*cos(th/2)^2
(%i147) (display2d:true,display(Amp_RL_RL),display2d:false)
                                               2 th
                              Amp_RL_RL = 2 cos (--)
                                                 2
```

We next look at the Dirac spinor normalization in the simplest case.

```
(%i148) " ======================================================="
(%i149) " in the high energy limit considered above, the spinor "
(%i150) " normalization becomes zero: "
(%i151) sbar(up1) . up1
(%o151) 0
(%i152) " In the arbitrary energy case, the normalization is 2*m "
(%i153) up1:UU(E,p,0,0,1)
(%i154) sbar(up1) . up1
(%o154) 2*sqrt(E-p)*sqrt(E+p)
(%i155) rootscontract(%)
(%o155) 2*sqrt(E^2-p^2)
(%i156) ratsubst(m,sqrt(E^2-p^2),%)
(%o156) 2*m
```

We can check the square of the polarized amplitude calculated above by using either the **mat_trace** method, the **m_tr** method, or the purely symbolic **nc_tr** method. The matrix method using **m_tr** syntax is designed to have arguments which look the same as the purely symbolic case.

```
(%i157) " ======================================================="
(%i158) " Using m_tr, mat_trace, and nc_tr for |A_RL_RL|^2 "
(%i159) " -----------------------------------------------------"
(%i160) " First, the mat_trace method using Dirac matrices."
(%i161) " P(sv) is the massless case helicity projection matrix "
(%i162) " which becomes (I4 + sv*Gam[5])/2."
(%i163) " In the massless limit P(+1) picks out negative helicty"
(%i164) " in the case of antiparticles."
(%i165) Mn_sq:(a12:mat_trace(P(1) . sL(p2)
                             . Gam[mu] . P(1) . sL(p1) . Gam[nu]),
               a34:mat_trace(P(1) . sL(p3)
                             . Gam[mu] . P(1) . sL(p4) . Gam[nu]),
               mcon(a12*a34,mu,nu),factor(%%))
(%o165) 16*(cos(th)+1)^2*E^4
(%i166) time(%)
(%o166) [0.05]
(%i167) Mn_sq:to_ao2(Mn_sq,th)
(%o167) 64*cos(th/2)^4*E^4
```

```
(%i168) M_sq:Mn_sq/s_th^2
(%o168) 4*cos(th/2)^4
(%i169) M_sq-Amp_RL_RL^2
(%o169) 0
(%i170) " which shows agreement with the explicit Dirac spinor method"
(%i171) " ------------------------------------------------------"
(%i172) "  Next the m_tr method, which is translated into mat_trace "
(%i173) Mn_sq:(a12:m_tr(S(1),p2,mu,S(1),p1,nu),
               a34:m_tr(S(1),p3,mu,S(1),p4,nu),mcon(a12*a34,mu,nu),factor(%%))
(%o173) 16*(cos(th)+1)^2*E^4
(%i174) time(%)
(%o174) [0.06]
(%i175) Mn_sq:to_ao2(Mn_sq,th)
(%o175) 64*cos(th/2)^4*E^4
(%i176) M_sq:Mn_sq/s_th^2
(%o176) 4*cos(th/2)^4
(%i177) M_sq-Amp_RL_RL^2
(%o177) 0
(%i178) " Finally the pure symbolic method, using nc_tr "
(%i179) " After using nc_tr =  noncov (tr(..)), contractions"
(%i180) " should be done with econ."
(%i181) " S(sv) --> (1 + sv*G5)/2, effectively, is the massless"
(%i182) " helicity projection operator for the symbolic method."
(%i183) Mn_sq:(a12:nc_tr(S(1),p2,mu,S(1),p1,nu),
               a34:nc_tr(S(1),p3,mu,S(1),p4,nu),econ(a12*a34,mu,nu),
               factor(%%))
(%o183) -16*(sin(th)^2-2*cos(th)-2)*E^4
(%i184) time(%)
(%o184) [0.14]
(%i185) Mn_sq:to_ao2(Mn_sq,th)
(%o185) 64*cos(th/2)^4*E^4
(%i186) M_sq:Mn_sq/s_th^2
(%o186) 4*cos(th/2)^4
(%i187) M_sq-Amp_RL_RL^2
(%o187) 0
```

Examples of symbolic traces of the product of **G5** with six or eight gammas.

```
(%i188) " ======================================================"
(%i189) " Traces of G5 with 6 and 8 gamma matrices "
(%i190) " Easier to follow if we use numbered Lorentz indices "
(%i191) index(n1,n2,n3,n4,n5,n6,n7,n8,n9,n10)
(%o191) [n1,n2,n3,n4,n5,n6,n7,n8,n9,n10]
(%i192) indexL
(%o192) [la,mu,nu,rh,si,ta,al,be,ga,n1,n2,n3,n4,n5,n6,n7,n8,n9,n10]
(%i193) tr6:tr(G5,n1,n2,n3,n4,n5,n6)
(%o193) 4*%i*Eps(n3,n2,n1,n4)*Gm(n5,n6)-4*%i*Eps(n3,n2,n1,n5)*Gm(n4,n6)
                                       +4*%i*Eps(n3,n2,n1,n6)*Gm(n4,n5)
                                       -4*%i*Gm(n1,n2)*Eps(n3,n4,n5,n6)
                                       +4*%i*Gm(n1,n3)*Eps(n2,n4,n5,n6)
                                       -4*%i*Eps(n1,n4,n5,n6)*Gm(n2,n3)

(%i194) tr8:tr(G5,n1,n2,n3,n4,n5,n6,n7,n8)
(%o194) 4*%i*Gm(n1,n2)*Eps(n5,n4,n3,n6)*Gm(n7,n8)
         -4*%i*Gm(n1,n3)*Eps(n5,n4,n2,n6)*Gm(n7,n8)
         +4*%i*Gm(n1,n4)*Eps(n5,n3,n2,n6)*Gm(n7,n8)
         -4*%i*Gm(n2,n3)*Eps(n5,n1,n4,n6)*Gm(n7,n8)
         +4*%i*Gm(n2,n4)*Eps(n5,n1,n3,n6)*Gm(n7,n8)
         -4*%i*Gm(n3,n4)*Eps(n5,n1,n2,n6)*Gm(n7,n8)
         +4*%i*Gm(n2,n3)*Eps(n4,n1,n6,n5)*Gm(n7,n8)
         +4*%i*Eps(n2,n1,n6,n5)*Gm(n3,n4)*Gm(n7,n8)
         -4*%i*Gm(n2,n4)*Eps(n3,n1,n6,n5)*Gm(n7,n8)
```

```
         +4*%i*Gm(n1,n5)*Eps(n2,n3,n4,n6)*Gm(n7,n8)
         -4*%i*Gm(n1,n6)*Eps(n2,n3,n4,n5)*Gm(n7,n8)
         -4*%i*Gm(n1,n2)*Eps(n5,n4,n3,n7)*Gm(n6,n8)
         +4*%i*Gm(n1,n3)*Eps(n5,n4,n2,n7)*Gm(n6,n8)
         -4*%i*Gm(n1,n4)*Eps(n5,n3,n2,n7)*Gm(n6,n8)
         +4*%i*Gm(n2,n3)*Eps(n5,n1,n4,n7)*Gm(n6,n8)
         -4*%i*Gm(n2,n4)*Eps(n5,n1,n3,n7)*Gm(n6,n8)
         +4*%i*Gm(n3,n4)*Eps(n5,n1,n2,n7)*Gm(n6,n8)
         -4*%i*Gm(n2,n3)*Eps(n4,n1,n7,n5)*Gm(n6,n8)
         -4*%i*Eps(n2,n1,n7,n5)*Gm(n3,n4)*Gm(n6,n8)
         +4*%i*Gm(n2,n4)*Eps(n3,n1,n7,n5)*Gm(n6,n8)
         -4*%i*Gm(n1,n5)*Eps(n2,n3,n4,n7)*Gm(n6,n8)
         +4*%i*Gm(n1,n7)*Eps(n2,n3,n4,n5)*Gm(n6,n8)
         +4*%i*Gm(n1,n2)*Eps(n5,n4,n3,n8)*Gm(n6,n7)
         -4*%i*Gm(n1,n3)*Eps(n5,n4,n2,n8)*Gm(n6,n7)
         +4*%i*Gm(n1,n4)*Eps(n5,n3,n2,n8)*Gm(n6,n7)
         -4*%i*Gm(n2,n3)*Eps(n5,n1,n4,n8)*Gm(n6,n7)
         +4*%i*Gm(n2,n4)*Eps(n5,n1,n3,n8)*Gm(n6,n7)
         -4*%i*Gm(n3,n4)*Eps(n5,n1,n2,n8)*Gm(n6,n7)
         +4*%i*Gm(n2,n3)*Eps(n4,n1,n8,n5)*Gm(n6,n7)
         +4*%i*Eps(n2,n1,n8,n5)*Gm(n3,n4)*Gm(n6,n7)
         -4*%i*Gm(n2,n4)*Eps(n3,n1,n8,n5)*Gm(n6,n7)
         +4*%i*Gm(n1,n5)*Eps(n2,n3,n4,n8)*Gm(n6,n7)
         -4*%i*Gm(n1,n8)*Eps(n2,n3,n4,n5)*Gm(n6,n7)
         -4*%i*Gm(n2,n3)*Eps(n4,n1,n6,n7)*Gm(n5,n8)
         -4*%i*Eps(n2,n1,n6,n7)*Gm(n3,n4)*Gm(n5,n8)
         +4*%i*Gm(n2,n4)*Eps(n3,n1,n6,n7)*Gm(n5,n8)
         +4*%i*Gm(n1,n6)*Eps(n2,n3,n4,n7)*Gm(n5,n8)
         -4*%i*Gm(n1,n7)*Eps(n2,n3,n4,n6)*Gm(n5,n8)
         +4*%i*Gm(n2,n3)*Eps(n4,n1,n6,n8)*Gm(n5,n7)
         +4*%i*Eps(n2,n1,n6,n8)*Gm(n3,n4)*Gm(n5,n7)
         -4*%i*Gm(n2,n4)*Eps(n3,n1,n6,n8)*Gm(n5,n7)
         -4*%i*Gm(n1,n6)*Eps(n2,n3,n4,n8)*Gm(n5,n7)
         +4*%i*Gm(n1,n8)*Eps(n2,n3,n4,n6)*Gm(n5,n7)
         -4*%i*Gm(n2,n3)*Eps(n4,n1,n7,n8)*Gm(n5,n6)
         -4*%i*Eps(n2,n1,n7,n8)*Gm(n3,n4)*Gm(n5,n6)
         +4*%i*Gm(n2,n4)*Eps(n3,n1,n7,n8)*Gm(n5,n6)
         +4*%i*Gm(n1,n7)*Eps(n2,n3,n4,n8)*Gm(n5,n6)
         -4*%i*Gm(n1,n8)*Eps(n2,n3,n4,n7)*Gm(n5,n6)
         +4*%i*Gm(n1,n2)*Gm(n3,n5)*Eps(n4,n6,n7,n8)
         -4*%i*Gm(n1,n3)*Gm(n2,n5)*Eps(n4,n6,n7,n8)
         +4*%i*Gm(n1,n5)*Gm(n2,n3)*Eps(n4,n6,n7,n8)
         -4*%i*Gm(n1,n2)*Eps(n3,n6,n7,n8)*Gm(n4,n5)
         +4*%i*Gm(n1,n3)*Eps(n2,n6,n7,n8)*Gm(n4,n5)
         -4*%i*Eps(n1,n6,n7,n8)*Gm(n2,n3)*Gm(n4,n5)
         +4*%i*Gm(n1,n4)*Gm(n2,n5)*Eps(n3,n6,n7,n8)
         -4*%i*Gm(n1,n5)*Gm(n2,n4)*Eps(n3,n6,n7,n8)
         -4*%i*Gm(n1,n4)*Eps(n2,n6,n7,n8)*Gm(n3,n5)
         +4*%i*Eps(n1,n6,n7,n8)*Gm(n2,n4)*Gm(n3,n5)
         +4*%i*Gm(n1,n5)*Eps(n2,n6,n7,n8)*Gm(n3,n4)
         -4*%i*Eps(n1,n6,n7,n8)*Gm(n2,n5)*Gm(n3,n4)

(%i195) tr10:tr(G5,n1,n2,n3,n4,n5,n6,n7,n8,n9,n10)
(%o195) g5TRACE(G5,n1,n2,n3,n4,n5,n6,n7,n8,n9,n10)
(%i196) " present reduce_g5 punts with ten gammas "
(%i197) " (homework problem?) "
```

12.5 moller0.mac: Scattering of Identical Scalar Charged Particles

The batch file **moller0.mac** treats the scattering event

$$\pi^+(p_1) + \pi^+(p_2) \to \pi^+(p_3) + \pi^+(p_4) \tag{12.23}$$

With the definitions

$$s = (p_1 + p_2)^2, \quad t = (p_1 - p_3)^2, \quad u = (p_1 - p_4)^2, \tag{12.24}$$

The symbolic method is used to derive the unpolarized differential cross section, starting with the invariant amplitude $M = M_1 + M_2$, in which

$$M_1 = -\frac{e^2\,(p_1 + p_3)\cdot(p_2 + p_4)}{t}, \tag{12.25}$$

and

$$M_2 = -\frac{e^2\,(p_1 + p_4)\cdot(p_2 + p_3)}{u}. \tag{12.26}$$

The Feynman rules for scalar electrodynamics are discussed in: G/R, QED, p. 434 ff, Renton, p.180 ff, I/Z, p. 282 ff, B/D, RQM, p. 195, Aitchison, p. 51 ff, A/H, p. 158 ff, Kaku, p. 158 ff, Quigg, p. 49, Schwinger, p. 284 ff, H/M, Ch. 4.

The list **invarR** (rules for invariant dot products) is first populated using the Dirac package function **invar**, which will be used by **tr** to replace invariant dot products such as **D(p1,p1)**, **D(p1,p2)**, etc by expressions in terms of the mass **m** of the scalar particle, and the Mandelstam variables **s, t, u**.

We then use **comp_def** to define the components of the 4-vectors in the center of momentum (CM) frame, and evaluate the Mandelstam variables in that frame.
Some othe Dirac package defined functions used in **moller0.mac** are **ev_Ds**, **pullfac**, **sub_stu**, and **fr_ao2**.

The batch file moller0.mac is used after loading in the dirac package. Both jobs are done using work.mac, which looks like:

```
/* work4/work.mac     */

  load ("dirac.mac")$
  batch ("moller0.mac")$
```

Inside Maxima, we then load in work.mac:

```
(%i1) load(work)$
                              work4 dirac.mac
                                   dgexp
                                   dgcon
                                  dgtrace
                                  dgeval
                                  dgmatrix
massL = [m,M]
indexL = [la,mu,nu,rh,si,ta,al,be,ga]
scalarL = []
" reserved program capital letter name use: "
" Chi, Con, D, Eps, G, G5, Gam, Gm, LI, Nlist, Nlast, UI, P, S, Sig"
"  UU, VP, VV, I2, Z2, CZ2, I4, Z4, CZ4, RZ4, N1,N2,... "

read and interpret file: #pc:/work4/moller0.mac
```

```
(%i2) " ======================================================"
(%i3) "  file moller0.mac "
(%i4) "  Maxima by Example, Ch. 12 "
(%i5) "  Dirac Algebra and Quantum Electrodynamics "
(%i6) "  Edwin L Woollett, woollett@charter.net "
(%i7) "  http://www.csulb.edu/~woollett  "
(%i8) print("      ver: ",_binfo%,"  date: ",mydate)
      ver:  Maxima 5.21.1   date:  2010-09-01

(%i9) " ======================================================"
(%i10) " ELASTIC SCATTERING OF SPIN 0 PARTICLES (SAME CHARGE)"
(%i11) " FOR EXAMPLE: "
(%i12) "      PI(+,p1) PI(+,p2)  --->  PI(+,p3) PI(+,p4)       "
(%i13) " POPULATE THE LIST invarR OF 4-VEC DOT PRODUCT VALUES, "
(%i14) " Using p1 + p2 = p3 + p4, s = (p1+p2)^2 = (p3+p4)^2 ,"
(%i15) "  t = (p1-p3)^2 = (p2-p4)^2,"
(%i16) "   and u = (p1-p4)^2 = (p2-p3)^2 "
(%i17) " ---------------------------------------------"
(%i18) invar(D(p1,p1) = m^2,D(p1,p2) = s/2-m^2,D(p1,p3) = m^2-t/2,
             D(p1,p4) = m^2-u/2,D(p2,p2) = m^2,D(p2,p3) = m^2-u/2,
             D(p2,p4) = m^2-t/2,D(p3,p3) = m^2,D(p3,p4) = s/2-m^2,
             D(p4,p4) = m^2)
(%i19) "------------------------------------------------------"
(%i20) " factor out -e^2 from Mfi, leaving Mfi = M1 + M2 "
(%i21) M1:D(p3+p1,p4+p2)/D(p1-p3,p1-p3)
(%i22) M2:D(p4+p1,p3+p2)/D(p1-p4,p1-p4)
(%i23) M1:ev_Ds(M1)
(%o23) s/t-u/t
(%i24) M1:pullfac(M1,1/t)
(%o24) (s-u)/t
(%i25) M2:ev_Ds(M2)
(%o25) s/u-t/u
(%i26) M2:pullfac(M2,1/u)
(%o26) (s-t)/u
(%i27) Mfi:M2+M1
(%o27) (s-u)/t+(s-t)/u
(%i28) " we get the result Mfi = (s-u)/t + (s-t)/u  "
(%i29) "CM FRAME EVALUATION "
(%i30) assume(p > 0,th >= 0,th <= %pi)
(%i31) comp_def(p1(E,0,0,p),p2(E,0,0,-p),p3(E,p*sin(th),0,p*cos(th)),
                p4(E,-p*sin(th),0,-p*cos(th)))
(%i32) s_th:noncov(D(p2+p1,p2+p1))
(%o32) 4*E^2
(%i33) t_th:factor(noncov(D(p1-p3,p1-p3)))
(%o33) 2*p^2*(cos(th)-1)
(%i34) u_th:factor(noncov(D(p1-p4,p1-p4)))
(%o34) -2*p^2*(cos(th)+1)
(%i35) Mfi:sub_stu(Mfi)
(%o35) -4*E^2/(p^2*sin(th)^2)-4/sin(th)^2+2
(%i36) " We want to combine the first two terms "
(%i37) " and replace p^2 by E^2 - m^2 in the numerator"
(%i38) Mfi_12:ratsimp(take_parts(Mfi,1,2))
(%o38) -(4*E^2+4*p^2)/(p^2*sin(th)^2)
(%i39) Mfi_12n:num(Mfi_12)
(%o39) -4*E^2-4*p^2
(%i40) Mfi_12n:factor(expand(subst(p^2 = E^2-m^2,Mfi_12n)))
(%o40) -4*(2*E^2-m^2)
(%i41) Mfi_12:Mfi_12n/denom(Mfi_12)
(%o41) -4*(2*E^2-m^2)/(p^2*sin(th)^2)
(%i42) " We now add in the third term and extract -2 "
(%i43) Mfi:pullfac(part(Mfi,3)+Mfi_12,-2)
(%o43) -2*(2*(2*E^2-m^2)/(p^2*sin(th)^2)-1)
```

```
(%i44) " having absorbed e^4 into a factor A which multiplies"
(%i45) " |Mfi|^2, and using e^2 = 4*%pi*alpha,"
(%i46) " in general, A = alpha^2*(pf/pi)/(4*Ecm^2) "
(%i47) " but here, pf = pi, and Ecm = 2*E, so "
(%i48) A:alpha^2/(16*E^2)
(%i49) " We can now write down the differential scattering cross section:"
(%i50) dsigdo:A*Mfi^2
(%i51) (display2d:true,display(dsigdo),display2d:false)
                                                  2    2
                                         2   2 (2 E  - m )      2
                                    alpha   (------------- - 1)
                                                2    2
                                               p  sin (th)
                           dsigdo = ---------------------------
                                                  2
                                              4 E

(%i52) " which agrees with Itzykson and Zuber, p. 286."
(%i53) " The factor 1/sin(th)^2 can be displayed in terms of th/2"
(%i54) " using: "
(%i55) fr_ao2(1/cos(th/2)^2+1/sin(th/2)^2,th)
(%o55) 4/sin(th)^2
(%i56) " and, of course p^2 = E^2 - m^2 "
```

Since this is the first example of using a batch file to program the calculations, we show here the appearance of the batch file **moller0.mac**.

To save space, we will only show the batch file run output for the remaining examples, even though the batch file output is rather compressed (minimum white space) and symbols are rearranged in the way Maxima likes to organize printed output.

Of course you can edit your own batch file run output to include more white space which would make the results more readable and easier to follow.

```
/* file moller0.mac
   pi(+) pi(+) --> pi(+) pi(+)
   in scalar electrodynamics, ie,
   ignoring structure of pions */

   /* references:
      Renton p.182
      I/Z p. 286
      Schwinger, pp. 285 - 289
      */
 " ======================================================"$
 "  file moller0.mac "$
 "  Maxima by Example, Ch. 12 "$
 "  Dirac Algebra and Quantum Electrodynamics "$
 "  Edwin L Woollett, woollett@charter.net "$
 "  http://www.csulb.edu/~woollett   "$
    print ("      ver: ",_binfo%,"  date: ",mydate )$
 " ======================================================"$
 " ELASTIC SCATTERING OF SPIN 0 PARTICLES (SAME CHARGE)"$
 " FOR EXAMPLE: "$
"      PI(+,p1) PI(+,p2)  --->  PI(+,p3) PI(+,p4)       "$
```

```
" POPULATE THE LIST invarR OF 4-VEC DOT PRODUCT VALUES, "$
" Using p1 + p2 = p3 + p4, s = (p1+p2)^2 = (p3+p4)^2 ,"$
"  t = (p1-p3)^2 = (p2-p4)^2,"$
"   and u = (p1-p4)^2 = (p2-p3)^2 "$
" ---------------------------------------------"$
invar (D(p1,p1) = m^2,
       D(p1,p2) = s/2 - m^2,
       D(p1,p3) = m^2 - t/2,
       D(p1,p4) = m^2 - u/2,
       D(p2,p2) = m^2,
       D(p2,p3) = m^2 - u/2,
       D(p2,p4) = m^2 - t/2,
       D(p3,p3) = m^2,
       D(p3,p4) = s/2 - m^2,
       D(p4,p4) = m^2)$
"---------------------------------------------------"$
" factor out -e^2 from Mfi, leaving Mfi = M1 + M2 "$

M1 : D(p1+p3,p2+p4)/D(p1-p3,p1-p3)$

M2 :  D(p1+p4,p2+p3)/D(p1-p4,p1-p4)$

M1 : ev_Ds (M1);

M1 : pullfac (M1,1/t);

M2 : ev_Ds (M2);

M2 : pullfac (M2,1/u);

Mfi : M1 + M2;

" we get the result Mfi = (s-u)/t + (s-t)/u  "$

"CM FRAME EVALUATION "$

assume ( p > 0, th >= 0, th <= %pi )$

comp_def ( p1( E,0,0,p),
           p2( E,0,0,-p),
           p3 (E,p*sin(th),0,p*cos(th)),
           p4 (E,-p*sin(th),0,-p*cos(th)) )$

s_th : noncov (D(p1+p2,p1+p2));

t_th : factor (noncov (D(p1-p3,p1-p3)));

u_th : factor (noncov (D(p1-p4,p1-p4)));
```

```
Mfi : sub_stu(Mfi);

" We want to combine the first two terms "$
" and replace p^2 by E^2 - m^2 in the numerator"$

Mfi_12 : ratsimp (take_parts (Mfi,1,2));

Mfi_12n : num (Mfi_12);

Mfi_12n : factor (expand (subst (p^2 = E^2 - m^2, Mfi_12n)));

Mfi_12 : Mfi_12n/denom(Mfi_12);

" We now add in the third term and extract -2 "$

Mfi : pullfac (Mfi_12 + part(Mfi,3),-2);

" having absorbed e^4 into a factor A which multiplies"$
" |Mfi|^2, and using e^2 = 4*%pi*alpha,"$
" in general, A = alpha^2*(pf/pi)/(4*Ecm^2) "$
" but here, pf = pi, and Ecm = 2*E, so "$

A : alpha^2/(16*E^2)$

" We can now write down the differential scattering cross section:"$

dsigdo : A*Mfi^2$

(display2d:true, display (dsigdo), display2d:false)$
" which agrees with Itzykson and Zuber, p. 286."$
" The factor 1/sin(th)^2 can be displayed in terms of th/2"$
" using: "$

fr_ao2(1/sin(th/2)^2 + 1/cos(th/2)^2, th );
" and, of course p^2 = E^2 - m^2 "$
```

12.6 moller1.mac: High Energy Elastic Scattering of Two Electrons

The batch file **moller1.mac** treats the high energy limit of the scattering event

$$e^-(p_1, \sigma_1) + e^-(p_2, \sigma_2) \longrightarrow e^-(p_3, \sigma_3) + e^-(p_4, \sigma_4) \tag{12.27}$$

We consider the unpolarized ultra-relativistic limit in which the masses of the electrons can be ignored ($|\mathbf{p}| >> m$).

Some references are: C. Moller, 1932, Schwinger, PSF I, p. 300 - 305, Greiner-Reinhardt, QED, 4'th, Sec. 3.3, Griffith Sec. 7.6, Jauch and Rohrlich, p. 252-257, Renton Sec. 4.3.1, B/D Sec. 7.9, BLP, Sec. 81.

(Note that BLP's symbol r_e should be replaced by α/m to compare with our results - they use (along with Greiner/Reinhardt) Gaussian units for electromagnetic equations, whereas we use (along with Peskin/Schroeder and Bjorken/Drell) rationalized Heaviside-Lorentz electromagnetic units. $e_{HL}^2 = 4\,\pi\,e_G^2$ See Greiner/Reinhardt, Sec 4.2.).

The invariant amplitude $M = M_1 - M_2$, has two terms. Using the definitions

$$s = (p_1 + p_2)^2, \quad t = (p_1 - p_3)^2, \quad u = (p_1 - p_4)^2, \tag{12.28}$$

and the abbreviations

$$u_1 = u(p_1, \sigma_1), \quad u_2 = u(p_2, \sigma_2), \quad \bar{u}_3 = \bar{u}(p_3, \sigma_3), \quad \bar{u}_4 = \bar{u}(p_4, \sigma_4) \tag{12.29}$$

we have

$$M_1 = -\frac{e^2\, \bar{u}_3\, \gamma^\mu\, u_1\, \bar{u}_4 \gamma_\mu\, u_2}{t}, \tag{12.30}$$

and

$$M_2 = -\frac{e^2\, \bar{u}_4\, \gamma^\mu\, u_1\, \bar{u}_3 \gamma_\mu\, u_2}{u}. \tag{12.31}$$

The unpolarized differential cross section in the CM frame is first found using symbolic methods, first in terms of **`s, t, u`** and then in terms of the scattering angle **`th`**.

The polarized amplitudes corresponding to definite electron helicity assignments are then calculated using explicit Dirac spinors and matrices, and the sum of the squares of these amplitudes reproduces the unpolarized symbolic calculation.

Finally one polarized squared amplitude is calculated using the symbolic methods.

```
(%i1) load(work)$
                                work4 dirac.mac
                                     dgexp
                                     dgcon
                                    dgtrace
                                    dgeval
                                    dgmatrix
massL = [m,M]
indexL = [la,mu,nu,rh,si,ta,al,be,ga]
scalarL = []
" reserved program capital letter name use: "
" Chi, Con, D, Eps, G, G5, Gam, Gm, LI, Nlist, Nlast, UI, P, S, Sig"
"  UU, VP, VV, I2, Z2, CZ2, I4, Z4, CZ4, RZ4, N1,N2,... "

read and interpret file: #pc:/work4/moller1.mac
(%i2) " ======================================================"
(%i3) "  file moller1.mac "
(%i4) "  Maxima by Example, Ch. 12 "
(%i5) "  Dirac Algebra and Quantum Electrodynamics "
(%i6) "  Edwin L Woollett, woollett@charter.net "
(%i7) "  http://www.csulb.edu/~woollett  "
(%i8) print("      ver: ",_binfo%,"  date: ",mydate)
      ver:  Maxima 5.21.1   date:  2010-09-01

(%i9) " ======================================================"
(%i10) "             MOLLER SCATTERING                   "
(%i11) " HIGH ENERGY LIMIT, CENTER OF MOMENTUM FRAME, NEGLECT MASSES "
(%i12) "  e(-,p1,sv1) + e(-,p2,sv2) --> e(-,p3,sv3) + e(-,p4,sv4)  "
(%i13) " m = electron mass is set to zero."
(%i14) " ------------------------------------------ "
(%i15) "NON-POLARIZED DIFFERENTIAL CROSS SECTION: SYMBOLIC METHODS"
(%i16) " POPULATE THE LIST invarR OF 4-VEC DOT PRODUCT VALUES, "
(%i17) " Using p1 + p2 = p3 + p4, s = (p1+p2)^2 = (p3+p4)^2 ,"
(%i18) "  t = (p1-p3)^2 = (p2-p4)^2,"
(%i19) "   and u = (p1-p4)^2 = (p2-p3)^2 "
```

```
(%i20) " CASE HIGH ENERGY (HE) LIMIT E >> m  "
(%i21) " ---------------------------------------------"
(%i22) invar(D(p1,p1) = 0,D(p1,p2) = s/2,D(p1,p3) = (-t)/2,D(p1,p4) = (-u)/2,
             D(p2,p2) = 0,D(p2,p3) = (-u)/2,D(p2,p4) = (-t)/2,D(p3,p3) = 0,
             D(p3,p4) = s/2,D(p4,p4) = 0)
(%i23) "------------------------------------------------------"
(%i24) " factor out -e^2 from Mfi, leaving Mfi = M1 - M2 "
(%i25) " With a sum over all helicities implied,"
(%i26) " |Mfi|^2 = M1n/t^2 + M2n/u^2 -M12n/(t*u) - M21n/(t*u) "
(%i27) " M1n = t^2 * M1*conj(M1), M2n = u^2 * M2*conj(M2) "
(%i28) " M12n = (t*u)*M1*conj(M2), M21n = (t*u)*M2*conj(M1), and: "
(%i29) M1n:factor(Con(tr(p3,mu,p1,nu)*tr(p4,mu,p2,nu),mu,nu))
(%o29) 8*(u^2+s^2)
(%i30) M2n:factor(Con(tr(p4,mu,p1,nu)*tr(p3,mu,p2,nu),mu,nu))
(%o30) 8*(t^2+s^2)
(%i31) " NOTE AUTOMATIC PRETRACE CONTRACTION OF REPEATED "
(%i32) " LORENTZ INDICES WITHIN A SINGLE TRACE OCCURS USING tr."
(%i33) M12n:factor(tr(p3,mu,p1,nu,p4,mu,p2,nu))
(%o33) -8*s^2
(%i34) M21n:factor(tr(p4,mu,p1,nu,p3,mu,p2,nu))
(%o34) -8*s^2
(%i35) MfiSQ:pullfac((-M21n)/(t*u)+(-M12n)/(t*u)+M2n/u^2+M1n/t^2,8)
(%o35) 8*((u^2+s^2)/t^2+2*s^2/(t*u)+(t^2+s^2)/u^2)
(%i36) " We have absorbed e^4 into A, with e^2 = 4*%pi*alpha "
(%i37) " Averaging over initial spins means we need to divide A by 4"
(%i38) " to get the unpolarized differential cross section (CM, HE)"
(%i39) A:alpha^2/(4*s)
(%i40) dsigdo_unpol_CM_HE:A*MfiSQ/4
(%o40) alpha^2*((u^2+s^2)/t^2+2*s^2/(t*u)+(t^2+s^2)/u^2)/(2*s)
(%i41) (display2d:true,display(dsigdo_unpol_CM_HE),display2d:false)
                                             2    2      2    2    2
                                         2  u  + s    2 s    t  + s
                                    alpha  (------- + ---- + -------)
                                                2     t u       2
                                               t               u
           dsigdo_unpol_CM_HE = ---------------------------------
                                                 2 s

(%i42) " which agrees with Renton's function of s,t, and u "
(%i43) " on page 159, Eq.(4.54) "
(%i44) " CONVERSION TO EXPLICIT FUNCTION OF SCATTERING ANGLE "
(%i45) assume(p > 0,th >= 0,th <= %pi)
(%i46) comp_def(p1(E,0,0,E),p2(E,0,0,-E),p3(E,E*sin(th),0,E*cos(th)),
                p4(E,-E*sin(th),0,-E*cos(th)))
(%i47) s_th:VP(p2+p1,p2+p1)
(%o47) 4*E^2
(%i48) t_th:factor(VP(p1-p3,p1-p3))
(%o48) 2*(cos(th)-1)*E^2
(%i49) u_th:factor(VP(p1-p4,p1-p4))
(%o49) -2*(cos(th)+1)*E^2
(%i50) " To get an expression we can compare with Renton and G/R, "
(%i51) " work with one term at a time "
(%i52) " sub_stu replaces s by s_th, t by t_th  "
(%i53) "  and u by u_th "
(%i54) M1SQ:sub_stu(M1n/t^2)
(%o54) 8*cos(th)^2/(cos(th)^2-2*cos(th)+1)+16*cos(th)/(cos(th)^2-2*cos(th)+1)
                                            +40/(cos(th)^2-2*cos(th)+1)
(%i55) " convert to (th/2) form "
(%i56) M1SQ:factor(ratsimp(to_ao2(M1SQ,th)))
(%o56) 8*(cos(th/2)^4+1)/sin(th/2)^4
(%i57) " ---------------- "
```

```
(%i58) M2SQ:sub_stu(M2n/u^2)
(%o58) 8*cos(th)^2/(cos(th)^2+2*cos(th)+1)-16*cos(th)/(cos(th)^2+2*cos(th)+1)
                                          +40/(cos(th)^2+2*cos(th)+1)
(%i59) M2SQ:ratsimp(to_ao2(M2SQ,th))
(%o59) (8*cos(th/2)^4-16*cos(th/2)^2+16)/cos(th/2)^4
(%i60) M2SQ:factor(ratsubst(1-sin(th/2)^2,cos(th/2)^2,num(M2SQ)))/denom(M2SQ)
(%o60) 8*(sin(th/2)^4+1)/cos(th/2)^4
(%i61) " ---------------------- "
(%i62) M12INT:-2*sub_stu(M12n/(t*u))
(%o62) -64/(cos(th)^2-1)
(%i63) M12INT:factor(M12INT)
(%o63) -64/((cos(th)-1)*(cos(th)+1))
(%i64) M12INT:subst([cos(th)-1 = -2*sin(th/2)^2,1+cos(th) = 2*cos(th/2)^2],
                    M12INT)
(%o64) 16/(cos(th/2)^2*sin(th/2)^2)
(%i65) " pull out a factor of 8 from each term. "
(%i66) MfiSQ:pullfac(M12INT+M2SQ+M1SQ,8)
(%o66) 8*((sin(th/2)^4+1)/cos(th/2)^4+2/(cos(th/2)^2*sin(th/2)^2)
                                    +(cos(th/2)^4+1)/sin(th/2)^4)
(%i67) dsigdo_unpol_CM_HE:A*MfiSQ/4
(%o67) alpha^2*((sin(th/2)^4+1)/cos(th/2)^4+2/(cos(th/2)^2*sin(th/2)^2)
                                          +(cos(th/2)^4+1)/sin(th/2)^4)
        /(2*s)
(%i68) (display2d:true,display(dsigdo_unpol_CM_HE),display2d:false)
                                4 th                                  4 th
                             sin (--) + 1                          cos (--) + 1
                         2        2                2                    2
                    alpha  (------------ + ---------------- + ------------)
                                  4 th          2 th     2 th          4 th
                               cos (--)      cos (--) sin (--)      sin (--)
                                    2             2        2             2
 dsigdo_unpol_CM_HE = -------------------------------------------------------
                                                2 s

(%i69) " which agrees with Renton, page 159, Eq.(4.55) "
(%i70) " as well as G/R, page 138, Eq. (3-139) "
(%i71) "-----------------------------------------------"
(%i72) " Convert to simple function of th displayed by G/R "
(%i73) MfiSQ:fr_ao2(MfiSQ,th)
(%o73) 16*cos(th)^4/sin(th)^4+96*cos(th)^2/sin(th)^4+144/sin(th)^4
(%i74) MfiSQ:factor(ratsimp(MfiSQ))
(%o74) 16*(cos(th)^2+3)^2/sin(th)^4
(%i75) dsigdo_unpol_CM_HE:A*MfiSQ/4
(%o75) alpha^2*(cos(th)^2+3)^2/(s*sin(th)^4)
(%i76) dsigdo_unpol_CM_HE:subst(s = s_th,dsigdo_unpol_CM_HE)
(%o76) alpha^2*(cos(th)^2+3)^2/(4*sin(th)^4*E^2)
(%i77) (display2d:true,display(dsigdo_unpol_CM_HE),display2d:false)
                                            2     2         2
                                       alpha  (cos (th) + 3)
                  dsigdo_unpol_CM_HE = ----------------------
                                                4      2
                                           4 sin (th) E

(%i78) " which agrees with G/R, page 139, top. "
```

The next section of **moller1.mac** uses explicit Dirac spinors and matrices to calculate polarized amplitudes.

```
(%i79) " ------------------------------------------------"
(%i80) " HE POLARIZED AMPLITUDES USING EXPLICIT DIRAC SPINORS "
(%i81) " ----------------------------------------------- "
(%i82) " case HE RR --> RR, ie, (+1,+1) --> (+1,+1) polarized amplitude "
(%i83) " Define the needed Dirac spinors and barred spinors."
(%i84) (up1:UU(E,E,0,0,1),up3b:sbar(UU(E,E,th,0,1)),up2:UU(E,E,%pi,0,1),
        up4b:sbar(UU(E,E,%pi-th,%pi,1)))
(%i85) " For example, following Halzen/Martin p. 120 ff :"
(%i86) " For helicity quantum numbers RR -> RR,  the amplitude Mfi"
(%i87) " (pulling out -e^2) has the form Mfi = M1 - M2 "
(%i88) " where M1 =  Mt/t and M2 =  Mu/u,  "
(%i89) " where t = (p1-p3)^2 and u = (p1-p4)^2, and "
(%i90) Mt:(a13:up3b . Gam[mu] . up1,a24:up4b . Gam[mu] . up2,mcon(a13*a24,mu),
           trigsimp(%%))
(%o90) -8*E^2
(%i91) Mu:(a14:up4b . Gam[mu] . up1,a23:up3b . Gam[mu] . up2,mcon(a14*a23,mu),
           trigsimp(%%))
(%o91) 8*E^2
(%i92) Mfi:Mt/t-Mu/u
(%o92) -8*E^2/u-8*E^2/t
(%i93) " Now replace s, t, and u by explicit functions of th "
(%i94) Mfi:sub_stu(Mfi)
(%o94) -8/(cos(th)^2-1)
(%i95) Mfi_RR_RR:ts(Mfi,th)
(%o95) 8/sin(th)^2
(%i96) " Now automate production of HE polarized amplitudes "
(%i97) " as functions of t and u "
(%i98) " Our definition of the amplitude does not include a factor of -e^2."
(%i99) he_me(sp1v,sp2v,sp3v,sp4v):=block([up1,up2,up3b,up4b,_mu%,Mt,Mu,temp],
              up1:UU(E,E,0,0,sp1v),up3b:sbar(UU(E,E,th,0,sp3v)),
              up2:UU(E,E,%pi,0,sp2v),up4b:sbar(UU(E,E,%pi-th,%pi,sp4v)),
              a13:up3b . Gam[_mu%] . up1,a24:up4b . Gam[_mu%] . up2,
              Mt:(Con(a13*a24,_mu%),expand(trigsimp(%%)))/t,
              a14:up4b . Gam[_mu%] . up1,a23:up3b . Gam[_mu%] . up2,
              Mu:(Con(a14*a23,_mu%),expand(trigsimp(%%)))/u,temp:Mt-Mu,
              if temp # 0 then temp:pullfac(temp,-8*E^2),temp)
(%i100) " test he_me for the case already worked out above "
(%i101) he_me(1,1,1,1)
(%o101) -8*(1/u+1/t)*E^2
(%i102) sub_stu(%)
(%o102) -8/(cos(th)^2-1)
(%i103) ts(%,th)
(%o103) 8/sin(th)^2
(%i104) " which agrees with our previous work."
(%i105) " ---------------------------------------- "
(%i106) " AN ALTERNATIVE PATH TO THE UNPOLARIZED CROSS SECTION "
(%i107) "  IS TO SUM THE ABSOLUTE VALUE SQUARED OF EACH "
(%i108) " OF THE POLARIZED AMPLITUDES, WHICH WE NOW DO."
(%i109) " The function Avsq(expr) is defined in dgmatrix.mac "
(%i110) " and computes the absolute value squared."
(%i111) "  (We could also use here abs(expr)^2 )"
(%i112) " Each nonzero amplitude is proportional to 8*E^2*e^2"
(%i113) " here since we have not replaced t or u in the denominators"
(%i114) " by a function of th."
(%i115) " We have already pulled out -e^2 from our 'amplitude' def"
```

```
(%i116) " Pull out also the factor 8*E^2 temporarily here."
(%i117) block([sL,s1,s2,s3,s4,temp],sL:[1,-1],mssq:0,print("  "),
            print(" svp1 svp2  svp3  svp4     amplitude     "),print("  "),
            for s1 in sL do
                for s2 in sL do
                    for s3 in sL do
                        for s4 in sL do
                            (temp:expand(he_me(s1,s2,s3,s4)/(8*E^2)),
                             mssq:Avsq(temp)+mssq,print("  "),
                             print(s1,s2,s3,s4," ",temp)),
            mssq:expand(fr_ao2(mssq,th)))
 svp1 svp2  svp3  svp4     amplitude

1 1 1 1   -1/u-1/t

1 1 1 -1   0

1 1 -1 1   0

1 1 -1 -1   0

1 -1 1 1   0

1 -1 1 -1   cos(th/2)^2/t

1 -1 -1 1   sin(th/2)^2/u

1 -1 -1 -1   0

-1 1 1 1   0

-1 1 1 -1   sin(th/2)^2/u

-1 1 -1 1   cos(th/2)^2/t

-1 1 -1 -1   0

-1 -1 1 1   0

-1 -1 1 -1   0

-1 -1 -1 1   0

-1 -1 -1 -1   -1/u-1/t
(%i118) " -------------------------------------- "
(%i119) mssq
(%o119) 4/(t*u)+cos(th)^2/(2*u^2)-cos(th)/u^2+5/(2*u^2)+cos(th)^2/(2*t^2)
             +cos(th)/t^2+5/(2*t^2)
(%i120) mssq:sub_stu(mssq)
(%o120) cos(th)^4/(4*sin(th)^4*E^4)+3*cos(th)^2/(2*sin(th)^4*E^4)
                                     +9/(4*sin(th)^4*E^4)
(%i121) mssq:factor(ratsimp(mssq))
(%o121) (cos(th)^2+3)^2/(4*sin(th)^4*E^4)
(%i122) " restore (8*E^2)^2 = 64*E^4 to mssq "
(%i123) mssq:64*E^4*mssq
(%o123) 16*(cos(th)^2+3)^2/sin(th)^4
(%i124) dsigdo_unpol_CM_HE:A*mssq/4
(%o124) alpha^2*(cos(th)^2+3)^2/(s*sin(th)^4)
(%i125) dsigdo_unpol_CM_HE:subst(s = s_th,%)
(%o125) alpha^2*(cos(th)^2+3)^2/(4*sin(th)^4*E^2)
```

```
(%i126) (display2d:true,display(dsigdo_unpol_CM_HE),display2d:false)
                                           2      2          2
                                      alpha   (cos  (th) + 3)
                  dsigdo_unpol_CM_HE = ----------------------
                                                4      2
                                          4 sin  (th) E

(%i127) " which agrees with the symbolic calculation."
```

The last section of **moller1.mac** uses symbolic methods to find the square of polarized amplitudes. An argument **S(1)** or **S(-1)** inside **tr** is effectively interpreted as $\frac{1}{2}\,(1 + \sigma\,\gamma^5)$, where $\sigma = \pm 1$ is the argument of **S**.

```
(%i128) " ================================  "
(%i129) " SYMBOLIC POLARIZED AMPLITUDE SQUARED "
(%i130) " ----------------------------------------"
(%i131) " Instead of summing over helicities, insert helicity "
(%i132) " projection operators appropriate to the zero mass limit "
(%i133) " CASE RR --> RR "
(%i134) M1n:trigsimp(Con(tr(S(1),p3,mu,S(1),p1,nu)*tr(S(1),p4,mu,S(1),p2,nu),
                 mu,nu))
(%o134) (-8*cos(th)^2-16*cos(th)+24)*E^4+2*u^2+2*s^2
(%i135) M2n:trigsimp(Con(tr(S(1),p4,mu,S(1),p1,nu)*tr(S(1),p3,mu,S(1),p2,nu),
                 mu,nu))
(%o135) (-8*cos(th)^2+16*cos(th)+24)*E^4+2*t^2+2*s^2
(%i136) " NOTE AUTOMATIC PRETRACE CONTRACTION OF REPEATED "
(%i137) " LORENTZ INDICES WITHIN A SINGLE TRACE OCCURS USING tr."
(%i138) M12n:tr(S(1),p3,mu,S(1),p1,nu,S(1),p4,mu,S(1),p2,nu)
(%o138) -4*s^2
(%i139) M21n:tr(S(1),p4,mu,S(1),p1,nu,S(1),p3,mu,S(1),p2,nu)
(%o139) -4*s^2
(%i140) MfiSQ:(-M21n)/(t*u)+(-M12n)/(t*u)+M2n/u^2+M1n/t^2
(%o140) ((-8*cos(th)^2+16*cos(th)+24)*E^4+2*t^2+2*s^2)/u^2
         +((-8*cos(th)^2-16*cos(th)+24)*E^4+2*u^2+2*s^2)/t^2+8*s^2/(t*u)
(%i141) MfiSQ:sub_stu(MfiSQ)
(%o141) 64/sin(th)^4
(%i142) "  which is the square of the Dirac spinor amplitude found above "
```

12.7 moller2.mac: Arbitrary Energy Moller Scattering

The batch file **moller2.mac** works out the case of arbitrary energy particles in the CM frame. See the references and kinematic notation in the previous section dealing with **moller1.mac**.

The first section of **moller2.mac** works out the unpolarized differential cross section for arbitrary energy, first in an arbitrary frame in terms of s, t, u, and then in the CM frame, using symbolic methods.

```
(%i1) load(work)$
                                work4 dirac.mac
                                     dgexp
                                     dgcon
                                    dgtrace
                                    dgeval
                                    dgmatrix
massL = [m,M]
indexL = [la,mu,nu,rh,si,ta,al,be,ga]
scalarL = []
" reserved program capital letter name use: "
" Chi, Con, D, Eps, G, G5, Gam, Gm, LI, Nlist, Nlast, UI, P, S, Sig"
"  UU, VP, VV, I2, Z2, CZ2, I4, Z4, CZ4, RZ4, N1,N2,... "
read and interpret file: #pc:/work4/moller2.mac
(%i2) " ====================================================="
```

```
(%i3) "  file moller2.mac "
(%i4) "  Maxima by Example, Ch. 12 "
(%i5) "  Dirac Algebra and Quantum Electrodynamics "
(%i6) "  Edwin L Woollett, woollett@charter.net "
(%i7) "  http://www.csulb.edu/~woollett  "
(%i8) print("      ver: ",_binfo%,"  date: ",mydate)
      ver:  Maxima 5.21.1   date:  2010-09-02

(%i9) "              MOLLER SCATTERING                   "
(%i10) "       ARBITRARY ENERGY, CENTER OF MOMENTUM FRAME "
(%i11) "  e(-,p1,sv1) + e(-,p2,sv2) --> e(-,p3,sv3) + e(-,p4,sv4) "
(%i12) " ------------------------------------------ "
(%i13) " SYMBOLIC TRACES FOR UNPOLARIZED DIFFERENTIAL CROSS SECTION "
(%i14) " Supply s, t, and u expressions for dot products "
(%i15) " ---------------------------------------------- "
(%i16) invar(D(p1,p1) = m^2,D(p1,p2) = s/2-m^2,D(p1,p3) = m^2-t/2,
             D(p1,p4) = m^2-u/2,D(p2,p2) = m^2,D(p2,p3) = m^2-u/2,
             D(p2,p4) = m^2-t/2,D(p3,p3) = m^2,D(p3,p4) = s/2-m^2,
             D(p4,p4) = m^2)
(%i17) M1n:Con(tr(m+p3,mu,m+p1,nu)*tr(m+p4,mu,m+p2,nu),mu,nu)
(%o17) 8*u^2-32*m^2*u+32*m^2*t+8*s^2-32*m^2*s+64*m^4
(%i18) M2n:Con(tr(m+p4,mu,m+p1,nu)*tr(m+p3,mu,m+p2,nu),mu,nu)
(%o18) 32*m^2*u+8*t^2-32*m^2*t+8*s^2-32*m^2*s+64*m^4
(%i19) " NOTE AUTOMATIC PRETRACE CONTRACTION OF REPEATED "
(%i20) " LORENTZ INDICES WITHIN A SINGLE TRACE OCCURS USING tr."
(%i21) M12n:tr(m+p3,mu,m+p1,nu,m+p4,mu,m+p2,nu)
(%o21) -16*m^2*u-16*m^2*t-8*s^2+48*m^2*s-32*m^4
(%i22) M21n:tr(m+p4,mu,m+p1,nu,m+p3,mu,m+p2,nu)
(%o22) -16*m^2*u-16*m^2*t-8*s^2+48*m^2*s-32*m^4
(%i23) " expressed as a function of s, t, and u "
(%i24) MfiSQ:(-M21n)/(t*u)+(-M12n)/(t*u)+M2n/u^2+M1n/t^2
(%o24) (8*u^2-32*m^2*u+32*m^2*t+8*s^2-32*m^2*s+64*m^4)/t^2
        +(32*m^2*u+8*t^2-32*m^2*t+8*s^2-32*m^2*s+64*m^4)/u^2
        +2*(16*m^2*u+16*m^2*t+8*s^2-48*m^2*s+32*m^4)/(t*u)
(%i25) " REPLACE s, t, and u WITH EXPLICIT FUNCTIONS OF th "
(%i26) " ------------------------------------------------ "
(%i27) assume(E > 0,p > 0,th >= 0,th <= %pi)
(%i28) comp_def(p1(E,0,0,p),p2(E,0,0,-p),p3(E,p*sin(th),0,p*cos(th)),
                p4(E,-p*sin(th),0,-p*cos(th)))
(%i29) s_th:VP(p2+p1,p2+p1)
(%o29) 4*E^2
(%i30) t_th:factor(VP(p1-p3,p1-p3))
(%o30) 2*p^2*(cos(th)-1)
(%i31) u_th:factor(VP(p1-p4,p1-p4))
(%o31) -2*p^2*(cos(th)+1)
(%i32) " ------------------------------------------------ "
(%i33) MfiSQ_th:factor(trigsimp(sub_stu(MfiSQ)))
(%o33) 16*(8*E^4+2*m^2*cos(th)^2*E^2-10*m^2*E^2+p^4*cos(th)^4+6*p^4*cos(th)^2
                +10*m^2*p^2*cos(th)^2+m^4*cos(th)^2+p^4-2*m^2*p^2+3*m^4)
        /(p^4*sin(th)^4)
(%i34) " show this equals BLP's expression in Eq. (81.10)"
(%i35) MfiSQ_cmp:16*(p^2+E^2)^2*((p^2/(p^2+E^2))^2*(4/sin(th)^2+1)
                                -3/sin(th)^2+4/sin(th)^4)
                 /p^4
(%i36) trigsimp(expand(MfiSQ_th-MfiSQ_cmp))
(%o36) ((48*sin(th)^2+64)*E^4+((96*p^2-32*m^2)*sin(th)^2-128*p^2-128*m^2)*E^2
                            +(-144*p^4-160*m^2*p^2-16*m^4)*sin(th)^2+64*p^4
                            +128*m^2*p^2+64*m^4)
        /(p^4*sin(th)^4)
(%i37) ratsubst(p^2+m^2,E^2,%)
(%o37) 0
(%i38) " which confirms equality."
```

```
(%i39) A:alpha^2/(4*s)
(%i40) A:sub_stu(A)
(%o40) alpha^2/(16*E^2)
(%i41) " Averaging over initial spins means we need to divide A by 4"
(%i42) " to get the unpolarized differential cross section "
(%i43) dsigdo_unpol_CM:A*MfiSQ_cmp/4
(%o43) alpha^2*(E^2+p^2)^2
              *(p^4*(4/sin(th)^2+1)/(E^2+p^2)^2-3/sin(th)^2+4/sin(th)^4)
        /(4*p^4*E^2)
(%i44) (display2d:true,display(dsigdo_unpol_CM),display2d:false)
                                       4     4
                                      p  (-------- + 1)
                                                 2
                          2    2    2 2      sin (th)          3          4
                     alpha  (E  + p )  (------------------ - --------- + --------)
                                             2    2 2           2          4
                                           (E  + p )        sin (th)   sin (th)
 dsigdo_unpol_CM = -----------------------------------------------------------
                                                 4  2
                                              4 p  E

(%i45) " which agrees with BLP, p. 323, Eq (81.10). "
(%i46) " ========================================================="
```

The next section of **`moller1.mac`** uses explicit Dirac spinors and matrices to calculate polarized amplitudes.

```
(%i47) " POLARIZED DIRAC SPINOR AMPLITUDES                        "
(%i48) " ---------------------------------------------------------"
(%i49) E_pm(expr):=expand(ratsubst(m^2+p^2,E^2,expr))
(%i50) p_Em(expr):=expand(ratsubst(E^2-m^2,p^2,expr))
(%i51) Ep_m(expr):=expand(ratsubst(m,sqrt(E-p)*sqrt(p+E),expr))
(%i52) Ep_Mm(expr):=(expand(ratsubst(M^2/4-m^2,p^2,expr)),
             expand(ratsubst(M/2,E,%%)))
(%i53) " -----------------------------------------------"
(%i54) " convert t_th and u_th  to th/2 forms "
(%i55) t_th2:to_ao2(t_th,th)
(%o55) -4*p^2*sin(th/2)^2
(%i56) u_th2:to_ao2(u_th,th)
(%o56) -4*p^2*cos(th/2)^2
(%i57) " dirac spinor amplitude given global s1,s2,s3,s4 "
(%i58) dA():=(
          (up1:UU(E,p,0,0,s1),up3b:sbar(UU(E,p,th,0,s3)),up2:UU(E,p,%pi,0,s2),
           up4b:sbar(UU(E,p,%pi-th,%pi,s4))),
          Mt:(a13:up3b . Gam[mu] . up1,a24:up4b . Gam[mu] . up2,
              mcon(a13*a24,mu)),Mt:Ep_m(Mt),M1:Mt/t_th2,
          Mu:(a14:up4b . Gam[mu] . up1,a23:up3b . Gam[mu] . up2,
              mcon(a14*a23,mu)),Mu:Ep_m(Mu),M2:Mu/u_th2,Mfi:M1-M2)
(%i59) " Print a table of polarized amplitudes."
(%i60) " Accumulate sum of squares mssq."
(%i61) block([sL,sv1,sv2,sv3,sv4,temp],sL:[1,-1],mssq:0,print("  "),
             print(" svp1 svp2  svp3  svp4     amplitude     "),print("  "),
             for sv1 in sL do
                 for sv2 in sL do
                     for sv3 in sL do
                         for sv4 in sL do
                             ([s1,s2,s3,s4]:[sv1,sv2,sv3,sv4],temp:dA(),
                              temp:E_pm(temp),mssq:Avsq(temp)+mssq,
                              print("  "),print(s1,s2,s3,s4," ",temp)),
             mssq:E_pm(mssq),mssq:expand(fr_ao2(mssq,th)))
```

```
 svp1 svp2  svp3  svp4     amplitude


1 1 1 1   m^2*sin(th/2)^2/(p^2*cos(th/2)^2)+2*sin(th/2)^2/cos(th/2)^2
                                 +m^2*cos(th/2)^2/(p^2*sin(th/2)^2)
                                 +2*cos(th/2)^2/sin(th/2)^2+4

1 1 1 -1   m*sin(th/2)*E/(p^2*cos(th/2))-m*cos(th/2)*E/(p^2*sin(th/2))

1 1 -1 1   m*sin(th/2)*E/(p^2*cos(th/2))-m*cos(th/2)*E/(p^2*sin(th/2))

1 1 -1 -1   2*m^2/p^2

1 -1 1 1   m*cos(th/2)*E/(p^2*sin(th/2))-m*sin(th/2)*E/(p^2*cos(th/2))

1 -1 1 -1   m^2*cos(th/2)^2/(p^2*sin(th/2)^2)+2*cos(th/2)^2/sin(th/2)^2-m^2/p^2

1 -1 -1 1   m^2*sin(th/2)^2/(p^2*cos(th/2)^2)+2*sin(th/2)^2/cos(th/2)^2-m^2/p^2

1 -1 -1 -1   m*sin(th/2)*E/(p^2*cos(th/2))-m*cos(th/2)*E/(p^2*sin(th/2))

-1 1 1 1   m*cos(th/2)*E/(p^2*sin(th/2))-m*sin(th/2)*E/(p^2*cos(th/2))

-1 1 1 -1   m^2*sin(th/2)^2/(p^2*cos(th/2)^2)+2*sin(th/2)^2/cos(th/2)^2-m^2/p^2

-1 1 -1 1   m^2*cos(th/2)^2/(p^2*sin(th/2)^2)+2*cos(th/2)^2/sin(th/2)^2-m^2/p^2

-1 1 -1 -1   m*sin(th/2)*E/(p^2*cos(th/2))-m*cos(th/2)*E/(p^2*sin(th/2))

-1 -1 1 1   2*m^2/p^2

-1 -1 1 -1   m*cos(th/2)*E/(p^2*sin(th/2))-m*sin(th/2)*E/(p^2*cos(th/2))

-1 -1 -1 1   m*cos(th/2)*E/(p^2*sin(th/2))-m*sin(th/2)*E/(p^2*cos(th/2))

-1 -1 -1 -1   m^2*sin(th/2)^2/(p^2*cos(th/2)^2)+2*sin(th/2)^2/cos(th/2)^2
                                 +m^2*cos(th/2)^2/(p^2*sin(th/2)^2)
                                 +2*cos(th/2)^2/sin(th/2)^2+4
(%i62) mssq
(%o62) -192*m^2/(p^2*sin(th)^2)-48*m^4/(p^4*sin(th)^2)-128/sin(th)^2
                               +256*m^2/(p^2*sin(th)^4)+64*m^4/(p^4*sin(th)^4)
                               +256/sin(th)^4+16
(%i63) " SHOW THIS IS THE SAME AS MfiSQ_th computed above with traces "
(%i64) MfiSQ_p:E_pm(MfiSQ_th)
(%o64) 16*cos(th)^4/sin(th)^4+192*m^2*cos(th)^2/(p^2*sin(th)^4)
                             +48*m^4*cos(th)^2/(p^4*sin(th)^4)
                             +96*cos(th)^2/sin(th)^4+64*m^2/(p^2*sin(th)^4)
                             +16*m^4/(p^4*sin(th)^4)+144/sin(th)^4
(%i65) trigsimp(mssq-MfiSQ_p)
(%o65) 0
(%i66) " which shows equality."
(%i67) " ========================================================"
```

12.8 bhabha1.mac: High Energy Limit of Bhabha Scattering

This batch file treats the high energy limit of the scattering event

$$e^-(p_1, \sigma_1) + e^+(p_2, \sigma_2) \rightarrow e^-(p_3, \sigma_3) + e^+(p_4, \sigma_4). \qquad (12.32)$$

This limit is that in which the masses of the particles can be ignored ($|\mathbf{p}| >> m$). In practice this means treating the leptons as massless particles.

Some references are: H.J. Bhabha, 1936, BLP QED, Sec. 81, B/D RQM, Sec. 7.9, G/R QED, Sec. 3.4, Renton EI, Sec. 4.3, and Schwinger, PSF I, p. 306 - 309.

The invariant amplitude $M = M_1 - M_2$ has two terms. Using the definitions

$$s = (p_1 + p_2)^2, \quad t = (p_1 - p_3)^2, \quad u = (p_1 - p_4)^2, \qquad (12.33)$$

and the abbreviations

$$u_1 = u(p_1, \sigma_1), \quad v_4 = v(p_4, \sigma_4), \quad \bar{u}_3 = \bar{u}(p_3, \sigma_3), \quad \bar{v}_2 = \bar{v}(p_2, \sigma_2) \qquad (12.34)$$

we have

$$M_1 = -\frac{e^2\, \bar{u}_3\, \gamma^\mu\, u_1\, \bar{v}_2 \gamma_\mu\, v_4}{t}, \qquad (12.35)$$

and

$$M_2 = -\frac{e^2\, \bar{v}_2\, \gamma^\mu\, u_1\, \bar{u}_3 \gamma_\mu\, v_4}{s}. \qquad (12.36)$$

The batch file **bhabha1.mac** uses the Dirac package functions **pullfac**, **VP**, **to_ao2**, **Avsq**, and **fr_ao2**.

The first section of **bhabha1.mac** derives the unpolarized differential cross section, first in an arbitrary frame in terms of s, t, u, and then in the CM frame in terms of the scattering angle **th**, using symbolic methods.

```
(%i1) load(work)$
                                 work4 dirac.mac
                                      dgexp
                                      dgcon
                                     dgtrace
                                     dgeval
                                     dgmatrix
massL = [m,M]
indexL = [la,mu,nu,rh,si,ta,al,be,ga]
scalarL = []
" reserved program capital letter name use: "
" Chi, Con, D, Eps, G, G5, Gam, Gm, LI, Nlist, Nlast, UI, P, S, Sig"
"  UU, VP, VV, I2, Z2, CZ2, I4, Z4, CZ4, RZ4, N1,N2,... "

read and interpret file: #pc:/work4/bhabha1.mac
(%i2) " ======================================================"
(%i3) "  file bhabha1.mac "
(%i4) "  Maxima by Example, Ch. 12 "
(%i5) "  Dirac Algebra and Quantum Electrodynamics "
(%i6) "  Edwin L Woollett, woollett@charter.net "
(%i7) "  http://www.csulb.edu/~woollett  "
(%i8) print("      ver: ",_binfo%,"  date: ",mydate)
      ver:  Maxima 5.21.1   date:  2010-09-02
```

```
(%i9) "              BHABHA SCATTERING                    "
(%i10) " HIGH ENERGY LIMIT, CENTER OF MOMENTUM FRAME, NEGLECT MASSES "
(%i11) "  e(-,p1,s1) + e(+,p2,s2) --> e(-,p3,s3) + e(+,p4,s4) "
(%i12) " m = electron and positron mass is set to zero."
(%i13) " ------------------------------------------ "
(%i14) "NON-POLARIZED DIFFERENTIAL CROSS SECTION: SYMBOLIC METHODS"
(%i15) " POPULATE THE LIST invarR OF 4-VEC DOT PRODUCT VALUES, "
(%i16) " Using p1 + p2 = p3 + p4, s = (p1+p2)^2 = (p3+p4)^2 ,"
(%i17) "  t = (p1-p3)^2 = (p2-p4)^2,"
(%i18) "   and u = (p1-p4)^2 = (p2-p3)^2 "
(%i19) " CASE HIGH ENERGY (HE) LIMIT E >> m  "
(%i20) " ---------------------------------------------"
(%i21) invar(D(p1,p1) = 0,D(p1,p2) = s/2,D(p1,p3) = (-t)/2,D(p1,p4) = (-u)/2,
             D(p2,p2) = 0,D(p2,p3) = (-u)/2,D(p2,p4) = (-t)/2,D(p3,p3) = 0,
             D(p3,p4) = s/2,D(p4,p4) = 0)
(%i22) "-----------------------------------------------------"
(%i23) " With a sum over all helicities implied,"
(%i24) " |Mfi|^2 = M1n/t^2 + M2n/s^2 -M12n/(t*s) - M21n/(t*s) "
(%i25) " M1n = t^2 * M1*conj(M1), M2n = s^2 * M2*conj(M2) "
(%i26) " M12n = (t*s)*M1*conj(M2), M21n = (t*s)*M2*conj(M1), and: "
(%i27) M1n:factor(Con(tr(p3,mu,p1,nu)*tr(p2,mu,p4,nu),mu,nu))
(%o27) 8*(u^2+s^2)
(%i28) M2n:factor(Con(tr(p2,mu,p1,nu)*tr(p3,mu,p4,nu),mu,nu))
(%o28) 8*(u^2+t^2)
(%i29) " NOTE AUTOMATIC PRETRACE CONTRACTION OF REPEATED "
(%i30) " LORENTZ INDICES WITHIN A SINGLE TRACE OCCURS USING tr."
(%i31) M12n:factor(tr(p3,mu,p1,nu,p2,mu,p4,nu))
(%o31) -8*u^2
(%i32) M21n:factor(tr(p2,mu,p1,nu,p3,mu,p4,nu))
(%o32) -8*u^2
(%i33) MfiSQ:pullfac((-M21n)/(t*s)+(-M12n)/(t*s)+M2n/s^2+M1n/t^2,8)
(%o33) 8*((u^2+t^2)/s^2+(u^2+s^2)/t^2+2*u^2/(s*t))
(%i34) " We have absorbed e^4 into A, with e^2 = 4*%pi*alpha "
(%i35) " Averaging over initial spins means we need to divide A by 4"
(%i36) " to get the unpolarized differential cross section (CM, HE)"
(%i37) A:alpha^2/(4*s)
(%i38) dsigdo_unpol_CM_HE:A*MfiSQ/4
(%o38) alpha^2*((u^2+t^2)/s^2+(u^2+s^2)/t^2+2*u^2/(s*t))/(2*s)
(%i39) (display2d:true,display(dsigdo_unpol_CM_HE),display2d:false)
                                           2    2    2    2      2
                                       2  u  + t    u  + s    2 u
                                  alpha  (------- + ------- + ----)
                                              2         2      s t
                                             s         t
          dsigdo_unpol_CM_HE = --------------------------------
                                               2 s

(%i40) " which agrees with Renton's function of s,t, and u "
(%i41) " on page 159, Eq.(4.54) "
(%i42) " CONVERSION TO EXPLICIT FUNCTION OF SCATTERING ANGLE "
(%i43) assume(E > 0,th >= 0,th <= %pi)
(%i44) comp_def(p1(E,0,0,E),p2(E,0,0,-E),p3(E,E*sin(th),0,E*cos(th)),
                 p4(E,-E*sin(th),0,-E*cos(th)))
(%i45) s_th:VP(p2+p1,p2+p1)
(%o45) 4*E^2
(%i46) t_th:factor(VP(p1-p3,p1-p3))
(%o46) 2*(cos(th)-1)*E^2
(%i47) u_th:factor(VP(p1-p4,p1-p4))
(%o47) -2*(cos(th)+1)*E^2
(%i48) "  CONVERT FROM s, t, u TO th form  "
(%i49) "--------------------------------------------------"
```

```
(%i50) MfiSQ_th:(sub_stu(MfiSQ),factor(%%))
(%o50) 4*(cos(th)^2+3)^2/(cos(th)-1)^2
(%i51) A_th:subst(s = s_th,A)
(%o51) alpha^2/(16*E^2)
(%i52) dsigdo_unpol_th:A_th*MfiSQ_th/4
(%o52) alpha^2*(cos(th)^2+3)^2/(16*(cos(th)-1)^2*E^2)
(%i53) (display2d:true,display(dsigdo_unpol_th),display2d:false)
                                          2      2          2
                                     alpha   (cos  (th) + 3)
                  dsigdo_unpol_th = ----------------------
                                                      2   2
                                      16 (cos(th) - 1)   E

(%i54) "------------------------------------------------------------"
(%i55) " which agrees with Renton, p. 160 "
```

The next section of **bhabha1.mac** uses explicit Dirac spinors and matrices to compute the polarized amplitudes.

```
(%i56) " HIGH ENERGY POLARIZED AMPLITUDES USING EXPLICIT DIRAC SPINORS "
(%i57) " ------------------------------------------------- "
(%i58) t_th2:to_ao2(t_th,th)
(%o58) -4*sin(th/2)^2*E^2
(%i59) " dirac spinor amplitude given global s1,s2,s3,s4 "
(%i60) dA():=(
          (up1:UU(E,E,0,0,s1),up3b:sbar(UU(E,E,th,0,s3)),
           vp2b:sbar(VV(E,E,%pi,0,s2)),vp4:VV(E,E,%pi-th,%pi,s4)),
          Mt:(a13:up3b . Gam[_mu%] . up1,a42:vp2b . Gam[_mu%] . vp4,
              mcon(a13*a42,_mu%),expand(trigsimp(%%))),M1:Mt/t_th2,
          Ms:(a12:vp2b . Gam[_mu%] . up1,a43:up3b . Gam[_mu%] . vp4,
              mcon(a12*a43,_mu%),expand(trigsimp(%%))),M2:Ms/s_th,M1-M2)
(%i61) " example: RR --> RR "
(%i62) [s1,s2,s3,s4]:[1,1,1,1]
(%i63) dA()
(%o63) 2/sin(th/2)^2
(%i64) " Make table of polarized amplitudes. "
(%i65) " Accumulate sum mssq of square of amplitudes."
(%i66) block([sL,sv1,sv2,sv3,sv4,temp],sL:[1,-1],mssq:0,print("  "),
             print(" s1 s2  s3  s4     amplitude     "),print("  "),
             for sv1 in sL do
                 for sv2 in sL do
                     for sv3 in sL do
                         for sv4 in sL do
                             ([s1,s2,s3,s4]:[sv1,sv2,sv3,sv4],temp:dA(),
                              mssq:Avsq(temp)+mssq,print("  "),
                              print(s1,s2,s3,s4,"        ",temp)),
             mssq:expand(fr_ao2(mssq,th)))

 s1 s2  s3  s4     amplitude

1 1 1 1          2/sin(th/2)^2

1 1 1 -1          0

1 1 -1 1          0

1 1 -1 -1          0

1 -1 1 1          0

1 -1 1 -1          2*cos(th/2)^2/sin(th/2)^2-2*cos(th/2)^2
```

```
1 -1 -1 1          -2*sin(th/2)^2

1 -1 -1 -1          0

-1 1 1 1          0

-1 1 1 -1          -2*sin(th/2)^2

-1 1 -1 1          2*cos(th/2)^2/sin(th/2)^2-2*cos(th/2)^2

-1 1 -1 -1          0

-1 -1 1 1          0

-1 -1 1 -1          0

-1 -1 -1 1          0

-1 -1 -1 -1          2/sin(th/2)^2
(%i67) " Sum of squares of polarized amplitudes:"
(%i68) mssq
(%o68) 4*cos(th)^4/(cos(th)^2-2*cos(th)+1)
        +24*cos(th)^2/(cos(th)^2-2*cos(th)+1)+36/(cos(th)^2-2*cos(th)+1)
(%i69) " COMPARE WITH SYMBOLIC RESULT MfiSQ_th CALCULATED ABOVE"
(%i70) trigsimp(mssq-MfiSQ_th)
(%o70) 0
(%i71) " WHICH SHOWS THEY ARE EQUIVALENT."
```

12.9 bhabha2.mac: Arbitrary Energy Bhabha Scattering

The batch file bhabha2.mac works out bhabha scattering for arbitrary particle energy in the CM frame. See the references and kinematic notation in the previous section dealing with **bhabha1.mac**. The batch file **bhabha1.mac** uses the Dirac package functions **pullfac**, **VP**, **sub_stu**, **to_ao2**, **Avsq**, and **fr_ao2**.

In addition, some kinematic substitution functions are defined in this batch file

- The function **E_pm(expr)** makes the replacement **E^2 --> m^2 + p^2**,
- The function **p_Em (expr)** makes the replacement **p^2 --> E^2 - m^2\mv**,
- The function **Ep_m (expr)** makes the replacement **sqrt (E-p)*sqrt (E+p) --> m**,
- The function **Ep_Mm (expr)** makes the two replacements: 1. **E --> M/2**, 2. **p^2 --> M^2/4 - m^2**, where **M** is the total center of momentum frame energy.

The first section of **bhabha2.mac** calculates the differential scattering cross section in the CM frame using symbolic methods.

```
(%i1) load(work)$
                                work4 dirac.mac
                                     dgexp
                                     dgcon
                                   dgtrace
                                   dgeval
                                   dgmatrix
massL = [m,M]
indexL = [la,mu,nu,rh,si,ta,al,be,ga]
scalarL = []
```

```
" reserved program capital letter name use: "
" Chi, Con, D, Eps, G, G5, Gam, Gm, LI, Nlist, Nlast, UI, P, S, Sig"
"  UU, VP, VV, I2, Z2, CZ2, I4, Z4, CZ4, RZ4, N1,N2,... "

read and interpret file: #pc:/work4/bhabha2.mac
(%i2) " ======================================================"
(%i3) "  file bhabha2.mac "
(%i4) "  Maxima by Example, Ch. 12 "
(%i5) "  Dirac Algebra and Quantum Electrodynamics "
(%i6) "  Edwin L Woollett, woollett@charter.net "
(%i7) "  http://www.csulb.edu/~woollett  "
(%i8) print("      ver: ",_binfo%,"  date: ",mydate)
      ver:  Maxima 5.21.1   date:  2010-09-02

(%i9) "              BHABHA SCATTERING                    "
(%i10) " Finite mass, Arbitrary Energy, CENTER OF MOMENTUM FRAME  "
(%i11) "  e(-,p1,s1) + e(+,p2,s2) --> e(-,p3,s3) + e(+,p4,s4) "
(%i12) " ---------------------------------------------- "
(%i13) " SYMBOLIC TRACES FOR UNPOLARIZED DIFFERENTIAL CROSS SECTION "
(%i14) " Supply s, t, and u expressions for dot products "
(%i15) " ---------------------------------------------------- "
(%i16) invar(D(p1,p1) = m^2,D(p1,p2) = s/2-m^2,D(p1,p3) = m^2-t/2,
             D(p1,p4) = m^2-u/2,D(p2,p2) = m^2,D(p2,p3) = m^2-u/2,
             D(p2,p4) = m^2-t/2,D(p3,p3) = m^2,D(p3,p4) = s/2-m^2,
             D(p4,p4) = m^2)
(%i17) " With a sum over all helicities implied,"
(%i18) " |Mfi|^2 = M1n/t^2 + M2n/s^2 -M12n/(t*s) - M21n/(t*s) "
(%i19) " M1n = t^2 * M1*conj(M1), M2n = s^2 * M2*conj(M2) "
(%i20) " M12n = (t*s)*M1*conj(M2), M21n = (t*s)*M2*conj(M1), and: "
(%i21) M1n:Con(tr(m+p3,mu,m+p1,nu)*tr(p2-m,mu,p4-m,nu),mu,nu)
(%o21) 8*u^2-32*m^2*u+32*m^2*t+8*s^2-32*m^2*s+64*m^4
(%i22) M2n:Con(tr(p2-m,mu,m+p1,nu)*tr(m+p3,mu,p4-m,nu),mu,nu)
(%o22) 8*u^2-32*m^2*u+8*t^2-32*m^2*t+32*m^2*s+64*m^4
(%i23) " NOTE AUTOMATIC PRETRACE CONTRACTION OF REPEATED "
(%i24) "  LORENTZ INDICES WITHIN A SINGLE TRACE OCCURS USING tr."
(%i25) M12n:tr(m+p3,mu,m+p1,nu,p2-m,mu,p4-m,nu)
(%o25) -8*u^2+48*m^2*u-16*m^2*t-16*m^2*s-32*m^4
(%i26) M21n:tr(p2-m,mu,m+p1,nu,m+p3,mu,p4-m,nu)
(%o26) -8*u^2+48*m^2*u-16*m^2*t-16*m^2*s-32*m^4
(%i27) " sum over spins of |Mfi|^2 in terms of s,t,u "
(%i28) MfiSQ:(-M21n)/(t*s)+(-M12n)/(t*s)+M2n/s^2+M1n/t^2
(%o28) (8*u^2-32*m^2*u+8*t^2-32*m^2*t+32*m^2*s+64*m^4)/s^2
        +(8*u^2-32*m^2*u+32*m^2*t+8*s^2-32*m^2*s+64*m^4)/t^2
        +2*(8*u^2-48*m^2*u+16*m^2*t+16*m^2*s+32*m^4)/(s*t)
(%i29) " replace s, t, u in terms of scatt. angle th "
(%i30) assume(E > 0,p > 0,th >= 0,th <= %pi)
(%i31) comp_def(p1(E,0,0,p),p2(E,0,0,-p),p3(E,p*sin(th),0,p*cos(th)),
                p4(E,-p*sin(th),0,-p*cos(th)))
(%i32) E_pm(expr):=expand(ratsubst(m^2+p^2,E^2,expr))
(%i33) s_th:VP(p2+p1,p2+p1)
(%o33) 4*E^2
(%i34) t_th:factor(VP(p1-p3,p1-p3))
(%o34) 2*p^2*(cos(th)-1)
(%i35) u_th:factor(VP(p1-p4,p1-p4))
(%o35) -2*p^2*(cos(th)+1)
(%i36) " ------------------------------------------------- "
(%i37) MfiSQ_th:sub_stu(MfiSQ)
(%i38) " replace E^2 by m^2 + p^2 "
(%i39) MfiSQ_p:E_pm(MfiSQ_th)
```

```
(%i40) MfiSQ_p:trigsimp(MfiSQ_p)
(%o40) (4*p^8*sin(th)^4+(-8*m^2*p^6*cos(th)-32*p^8-72*m^2*p^6-52*m^4*p^4)
                        *sin(th)^2+(32*m^2*p^6+80*m^4*p^4+56*m^6*p^2)*cos(th)
                       +64*p^8+160*m^2*p^6+128*m^4*p^4+40*m^6*p^2+16*m^8)
        /((p^8+2*m^2*p^6+m^4*p^4)*cos(th)^2
         +(-2*p^8-4*m^2*p^6-2*m^4*p^4)*cos(th)+p^8+2*m^2*p^6+m^4*p^4)
(%i41) " ------------------------------------------------"
(%i42) " compare to Greiner, Reinhardt expression p. 148 "
(%i43) GR1:(2*p^4*(cos(th/2)^4+1)+4*p^2*m^2*cos(th/2)^2+m^4)/(p^4*sin(th/2)^4)
(%o43) (2*p^4*(cos(th/2)^4+1)+4*m^2*p^2*cos(th/2)^2+m^4)/(p^4*sin(th/2)^4)
(%i44) GR2:(p^4*(cos(th)^2+1)+4*p^2*m^2+3*m^4)/E^4
(%o44) (p^4*(cos(th)^2+1)+4*m^2*p^2+3*m^4)/E^4
(%i45) GR3:(-(4*p^4*cos(th/2)^4+8*p^2*m^2*cos(th/2)^2+3*m^4))
            /(E^2*p^2*sin(th/2)^2)
(%o45) (-4*p^4*cos(th/2)^4-8*m^2*p^2*cos(th/2)^2-3*m^4)/(p^2*sin(th/2)^2*E^2)
(%i46) " replace th/2 and E^2 for comparison "
(%i47) GR1_r:fr_ao2(GR1,th)
(%i48) GR2_r:E_pm(GR2)
(%i49) GR3_r:E_pm(fr_ao2(GR3,th))
(%i50) GR:trigsimp(expand(GR3_r+GR2_r+GR1_r))
(%i51) trigsimp(GR-MfiSQ_p/4)
(%o51) 0
(%i52) " which shows the equivalence we need "
(%i53) " dividing MfiSQ by 4 comes from averaging over"
(%i54) " the initial helicities "
(%i55) " We have absorbed e^4 into A, with e^2 = 4*%pi*alpha "
(%i56) " to get the unpolarized differential cross section (CM)"
(%i57) A:alpha^2/(4*s_th)
(%i58) GR_expr:GR3+GR2+GR1
(%i59) dsigdo_unpol_CM:A*GR_expr
(%i60) (display2d:true,display(dsigdo_unpol_CM),display2d:false)
                                4    4 th        2  2    2 th       4
                           - 4 p  cos (--) - 8 m  p  cos (--) - 3 m
                        2               2                  2
dsigdo_unpol_CM = (alpha  (-----------------------------------------
                                          2    2 th   2
                                         p  sin (--) E
                                                  2
    4     2               2  2      4
   p  (cos (th) + 1) + 4 m  p  + 3 m
 + ---------------------------------
                    4
                   E
      4     4 th            2  2    2 th     4
   2 p  (cos (--) + 1) + 4 m  p  cos (--) + m
             2                      2             2
 + ------------------------------------------))/(16 E )
                    4    4 th
                   p  sin (--)
                            2
(%i61) " which agrees with G/R, Exer. 3.8, eqn. 21, p. 148 "
```

The next section of **bhabha2.mac** uses explicit Dirac spinors and matrices to compute polarized amplitudes.

```
(%i62) " ============================================================ "
(%i63) " POLARIZED DIRAC SPINOR AMPLITUDES                            "
(%i64) " ----------------------------------------------------------"
(%i65) p_Em(expr):=expand(ratsubst(E^2-m^2,p^2,expr))
(%i66) Ep_m(expr):=expand(ratsubst(m,sqrt(E-p)*sqrt(p+E),expr))
(%i67) Ep_Mm(expr):=(expand(ratsubst(M^2/4-m^2,p^2,expr)),
            expand(ratsubst(M/2,E,%%)))
(%i68) " ------------------------------------------------"
```

```
(%i69) " convert t_th to th/2 form "
(%i70) t_th2:to_ao2(t_th,th)
(%o70) -4*p^2*sin(th/2)^2
(%i71) " dirac spinor amplitude given global s1,s2,s3,s4 "
(%i72) dA():=(
           (up1:UU(E,p,0,0,s1),up3b:sbar(UU(E,p,th,0,s3)),
            vp2b:sbar(VV(E,p,%pi,0,s2)),vp4:VV(E,p,%pi-th,%pi,s4)),
          Mt:(a13:up3b . Gam[mu] . up1,a42:vp2b . Gam[mu] . vp4,
              mcon(a13*a42,mu),Ep_m(%%),E_pm(%%),trigsimp(%%)),M1:Mt/t_th2,
          Ms:(a12:vp2b . Gam[mu] . up1,a43:up3b . Gam[mu] . vp4,
              mcon(a12*a43,mu),Ep_m(%%),E_pm(%%),trigsimp(%%)),M2:Ms/s_th,
          M2-M1)
(%i73) " example: RR --> RR  "
(%i74) [s1,s2,s3,s4]:[1,1,1,1]
(%o74) [1,1,1,1]
(%i75) dA()
(%o75) (4*m^2-8*m^2*sin(th/2)^2)/(4*E^2)
        +(-4*m^2*cos(th/2)^2-8*p^2)/(4*p^2*sin(th/2)^2)
(%i76) " ============================================================ "
(%i77) " table of polarized amplitudes generates global mssq "
(%i78) block([sL,sv1,sv2,sv3,sv4,temp],sL:[1,-1],mssq:0,print("  "),
             print(" s1  s2  s3  s4         amplitude     "),print("  "),
             for sv1 in sL do
                 for sv2 in sL do
                     for sv3 in sL do
                         for sv4 in sL do
                             ([s1,s2,s3,s4]:[sv1,sv2,sv3,sv4],temp:dA(),
                              print("  "),print(s1,s2,s3,s4," ",temp),
                              temp:E_pm(temp),mssq:Avsq(temp)+mssq),
             mssq:E_pm(mssq),mssq:expand(fr_ao2(mssq,th)),mssq:trigsimp(mssq))

 s1  s2  s3  s4         amplitude

1 1 1 1   (4*m^2-8*m^2*sin(th/2)^2)/(4*E^2)+(-4*m^2*cos(th/2)^2-8*p^2)
                                  /(4*p^2*sin(th/2)^2)

1 1 1 -1   2*m*cos(th/2)*sin(th/2)/E-m*cos(th/2)*E/(p^2*sin(th/2))

1 1 -1 1   m*cos(th/2)*E/(p^2*sin(th/2))-2*m*cos(th/2)*sin(th/2)/E

1 1 -1 -1   (4*m^2-8*m^2*sin(th/2)^2)/(4*E^2)+m^2/p^2

1 -1 1 1   m*cos(th/2)*E/(p^2*sin(th/2))-2*m*cos(th/2)*sin(th/2)/E

1 -1 1 -1   (8*p^2+8*m^2)*cos(th/2)^2/(4*E^2)+(-8*p^2-4*m^2)*cos(th/2)^2
                                  /(4*p^2*sin(th/2)^2)

1 -1 -1 1   (8*p^2+8*m^2)*sin(th/2)^2/(4*E^2)-m^2/p^2

1 -1 -1 -1   m*cos(th/2)*E/(p^2*sin(th/2))-2*m*cos(th/2)*sin(th/2)/E

-1 1 1 1   2*m*cos(th/2)*sin(th/2)/E-m*cos(th/2)*E/(p^2*sin(th/2))

-1 1 1 -1   (8*p^2+8*m^2)*sin(th/2)^2/(4*E^2)-m^2/p^2

-1 1 -1 1   (8*p^2+8*m^2)*cos(th/2)^2/(4*E^2)+(-8*p^2-4*m^2)*cos(th/2)^2
                                  /(4*p^2*sin(th/2)^2)

-1 1 -1 -1   2*m*cos(th/2)*sin(th/2)/E-m*cos(th/2)*E/(p^2*sin(th/2))

-1 -1 1 1   (4*m^2-8*m^2*sin(th/2)^2)/(4*E^2)+m^2/p^2
```

```
-1 -1 1 -1   2*m*cos(th/2)*sin(th/2)/E-m*cos(th/2)*E/(p^2*sin(th/2))

-1 -1 -1 1   m*cos(th/2)*E/(p^2*sin(th/2))-2*m*cos(th/2)*sin(th/2)/E

-1 -1 -1 -1    (4*m^2-8*m^2*sin(th/2)^2)/(4*E^2)+(-4*m^2*cos(th/2)^2-8*p^2)
                                    /(4*p^2*sin(th/2)^2)
(%i79) "------------------------------------------"
(%i80) " sum of squares of polarized amplitudes:"
(%i81) mssq
(%o81) (4*p^8*sin(th)^4+(-8*m^2*p^6*cos(th)-32*p^8-72*m^2*p^6-52*m^4*p^4)
                         *sin(th)^2+(32*m^2*p^6+80*m^4*p^4+56*m^6*p^2)*cos(th)
                        +64*p^8+160*m^2*p^6+128*m^4*p^4+40*m^6*p^2+16*m^8)
        /((p^8+2*m^2*p^6+m^4*p^4)*cos(th)^2
         +(-2*p^8-4*m^2*p^6-2*m^4*p^4)*cos(th)+p^8+2*m^2*p^6+m^4*p^4)
(%i82) " SHOW THIS IS THE SAME AS MfiSQ_th computed above with traces "
(%i83) MfiSQ_p:E_pm(MfiSQ_th)
(%i84) trigsimp(mssq-MfiSQ_p)
(%o84) 0
(%i85) " which shows equality."
(%i86) " ============================================================= "
```

12.10 photon1.mac: Photon Transverse Polarization 3-Vector Sums

In this section we use Maxima to work out some well known properties of photon polarization 3-vectors corresponding to physical (external) photons, using the batch file **photon1.mac**. Some references to this subject are: G/R QED p. 187, Weinberg I p. 352, 360, 368, Renton p. 166, and Gross p. 315.

Let $\mathbf{e}_{\mathbf{k}\,\mathbf{s}}$ be a real photon polarization 3-vector which is transverse to the initial photon 3-momentum $\mathbf{k}$, with $s = 1$ being parallel to the scattering plane, and $s = 2$ being perpendicular to the scattering plane.

Let $\mathbf{e}_{\mathbf{k}'\,\mathbf{r}}$ be a real photon polarization 3-vector which is transverse to the final photon 3-momentum $\mathbf{k}'$ with $r = 1$ being parallel to the scattering plane, and $r = 2$ being perpendicular to the scattering plane. Then we have the relations

$$\sum_{\mathbf{s}=1}^{2} (\mathbf{e}_{\mathbf{k}\,\mathbf{s}} \cdot \mathbf{e}_{\mathbf{k}'\,\mathbf{r}})^2 = \mathbf{1} - \left(\hat{\mathbf{k}} \cdot \mathbf{e}_{\mathbf{k}'\,\mathbf{r}}\right)^2 \tag{12.37}$$

$$\sum_{\mathbf{s,r}=1}^{2} (\mathbf{e}_{\mathbf{k}\,\mathbf{s}} \cdot \mathbf{e}_{\mathbf{k}'\,\mathbf{r}})^2 = \mathbf{1} + \cos^2 \boldsymbol{\theta} \tag{12.38}$$

Here we are using real polarization 3-vectors to describe states of linear polarization.

```
(%i1) batch("photon1.mac")$
read and interpret file: #pC:/work4/photon1.mac
(%i2)                          display2d : false
(%i3) " ======================================================"
(%i4) "  file photon1.mac "
(%i5) "  Maxima by Example, Ch. 12 "
(%i6) "  Dirac Algebra and Quantum Electrodynamics "
(%i7) "  Edwin L Woollett, woollett@charter.net "
(%i8) "  http://www.csulb.edu/~woollett  "
(%i9) print("      ver: ",_binfo%,"  date: ",mydate)
      ver:  Maxima 5.21.1   date:  2010-09-02
(%i10) "------------------------------------------------------"
```

```
(%i11) "  PROPERTIES AND SUMS OF TRANSVERSE POLARIZATION VECTORS "
(%i12) " ---------------------------------------------------- "
(%i13) " define 3-vectors as lists "
(%i14) " scattering plane is z-x plane "
(%i15) k_vec:[0,0,k]
(%i16) kp_vec:[kp*sin(th),0,kp*cos(th)]
(%i17) " e[1] X e[2] = k_vec/k "
(%i18) e[1]:[1,0,0]
(%i19) e[1] . e[1]
(%o19) 1
(%i20) e[1] . k_vec
(%o20) 0
(%i21) e[2]:[0,1,0]
(%i22) e[2] . e[2]
(%o22) 1
(%i23) e[2] . e[1]
(%o23) 0
(%i24) e[2] . k_vec
(%o24) 0
(%i25) " ep[1] X ep[2] = kp_vec/kp "
(%i26) " Case: parallel to the scattering plane "
(%i27) ep[1]:[cos(th),0,-sin(th)]
(%i28) trigsimp(ep[1] . ep[1])
(%o28) 1
(%i29) ep[1] . kp_vec
(%o29) 0
(%i30) " Case: perpendicular to the scattering plane "
(%i31) ep[2]:[0,1,0]
(%i32) ep[2] . ep[2]
(%o32) 1
(%i33) ep[2] . ep[1]
(%o33) 0
(%i34) ep[2] . kp_vec
(%o34) 0
(%i35) "============================================"
(%i36) " photon transverse polarization sum over both leads to "
(%i37) "  (1 + cos(th)^2 ) "
(%i38) sum(sum((e[s] . ep[r])^2,s,1,2),r,1,2)
(%o38) cos(th)^2+1
(%i39) "============================================"
(%i40) " Sum over only e[1] and e[2] values leads to "
(%i41) " sum ( (e[s] . ep[r])^2,s,1,2) = 1 - (ku . ep[r])^2 "
(%i42) " where ku is unit vector along k_vec "
(%i43) ku:k_vec/k
(%o43) [0,0,1]
(%i44) " first for r = 1 "
(%i45) sum((e[s] . ep[1])^2,s,1,2)
(%o45) cos(th)^2
(%i46) trigsimp(1-(ku . ep[1])^2)
(%o46) cos(th)^2
(%i47) " next for r = 2 "
(%i48) sum((e[s] . ep[2])^2,s,1,2)
(%o48) 1
(%i49) 1-(ku . ep[2])^2
(%o49) 1
(%i50) "============================================"
```

12.11 Covariant Physical External Photon Polarization 4-Vectors - A Review

We will use the following relation for covariant physical photon polarization 4-vectors which are 4-orthogonal to the photon 4-momentum vector. This formalism (Sterman, p. 220, De Wit/Smith, p.141, Schwinger, p. 73 and pp. 291 - 294, Jauch/Rohrlich, p. 440) uses gauge freedom within the Lorentz gauge.

The two physical polarization states of real external photons corresond to $\lambda = 1, 2$, and for $\mu,\ \nu = 0, 1, 2, 3$,

$$P^{\mu\nu}(k) = \sum_{\lambda=1,2} e^{\mu}_{k\,\lambda}\, e^{\nu *}_{k\,\lambda} = -g^{\mu\nu} + \frac{(k^{\mu}\,\bar{k}^{\nu} + k^{\nu}\,\bar{k}^{\mu})}{k\cdot\bar{k}}, \tag{12.39}$$

where there exists a reference frame in which if $k^{\mu} = (k^0, \mathbf{k})$ then $\bar{k}^{\mu} = (k^0, -\mathbf{k})$.

Let n^{μ} be a unit timelike 4-vector which is 4-orthogonal to $e^{\mu}_{k\,\lambda}$. Then define

$$\bar{k}^{\mu} = -k^{\mu} + 2\,n^{\mu}\,(k\cdot n). \tag{12.40}$$

In an arbitrary frame, $k\cdot k = 0$, $\bar{k}\cdot\bar{k} = 0$, $k\cdot e_{k\,\lambda} = 0$, $\bar{k}\cdot e_{k\,\lambda} = 0$, $n\cdot e_{k\,\lambda} = 0$, $n\cdot n = +1$, $k\cdot\bar{k} = 2\,(k\cdot n)^2$ and

$$e^{*}_{k\,\lambda}\cdot e_{k\,\lambda'} = -\delta_{\lambda\,\lambda'}, \quad \lambda, \lambda' = 1, 2 \tag{12.41}$$

An alternative but less compact form of the physical photon polarization tensor $P^{\mu\nu}(k)$ is

$$P^{\mu\nu}(k) = \sum_{\lambda=1,2} e^{\mu}_{k\,\lambda}\, e^{\nu *}_{k\,\lambda} = -g^{\mu\nu} - \frac{k^{\mu}\,k^{\nu}}{(k\cdot n)^2} + \frac{(k^{\mu}\,n^{\nu} + k^{\nu}\,n^{\mu})}{k\cdot n} \tag{12.42}$$

We can always identify the unit timelike 4-vector n^{μ} with either the total 4-momentum vector (thus working in the CM frame) or the 4-momentum of a particular particle (thus working in the rest frame of that particle).

Thus if in a chosen frame, $\{p^{\mu}\} = \{m, 0, 0, 0\}$, we identify $n^{\mu} = p^{\mu}/m$, so that $\{n^{\mu}\} = \{1, 0, 0, 0\}$, then in that frame $\bar{k}^0 = k^0$ and $\bar{k}^j = -k^j$.

Then $n\cdot e_{k\,\lambda} = 0$ implies that $e^0_{k\,\lambda} = 0$, and then $k\cdot e_{k\,\lambda} = 0$ reduces to $\mathbf{k}\cdot\mathbf{e}_{k\,\lambda} = 0$ (for $\lambda = 1, 2$), and the two polarization 3-vectors $\mathbf{e}_{k\,\lambda}$ must be chosen perpendicular to $\mathbf{k}$ and to each other: $\mathbf{e}^{*}_{k\,1}\cdot\mathbf{e}_{k\,2} = 0$.

12.12 compton0.mac: Compton Scattering by a Spin 0 Structureless Charged Particle

In this section we use the batch file **`compton0.mac`** to work out details of Compton scattering of scalar charged particles. Some references to this subject can be found in G/R QED, p 435, B/D RQM, p. 193, Schwinger PSF I, p290, Aitchison RQM, p87, De Wit/Smith, p 147, I/Z p. 286.

Three diagrams contribute to Compton scattering of a scalar charged particle which we can for convenience think of as

$$\pi^{+}(p_1 + \gamma(k_1, \lambda_1) \rightarrow \pi^{+}(p_2 + \gamma(k_2, \lambda_2) \tag{12.43}$$

Let

$$s = (k_1 + p_1)^2, \quad t = (k_1 - k_2)^2, \quad u = (k_1 - p_2)^2. \tag{12.44}$$

Four momentum conservation means $p_1 + k_1 = p_2 + k_2$. If we ignore a factor $-e^2$ and use real photon polarization 4-vectors, the invariant amplitude is

$$M = e^{\mu}_{k_2\,\lambda_2}\left(2\,g_{\mu\nu} - \frac{(2\,p_2 + k_2)_{\mu}\,(2\,p_1 + k_1)_{\nu}}{s - m^2} - \frac{(2\,p_1 - k_2)_{\mu}\,(2\,p_2 - k_1)_{\nu}}{u - m^2}\right) e^{\nu}_{k_1\,\lambda_1} \tag{12.45}$$

Use of $k_1 \cdot e_{k_1\,\lambda_1} = 0$, $k_2 \cdot e_{k_2\,\lambda_2} = 0$, $s - m^2 = 2\,p_1 \cdot k_1$, and $u - m^2 = -2\,p_2 \cdot k_1$ leads to

$$M = 2\,e^{\mu}_{k_2\,\lambda_2}\,T_{\mu\,\nu}\,e^{\nu}_{k_1\,\lambda_1} \tag{12.46}$$

where

$$T_{\mu\,\nu} = g_{\mu\,\nu} - \frac{p_{2\,\mu}\,p_{1\,\nu}}{k_1 \cdot p_1} + \frac{p_{2\,\nu}\,p_{1\,\mu}}{k_1 \cdot p_2} \tag{12.47}$$

We form the absolute square of M and sum over λ_2 first,

$$\sum_{\lambda_2} |M|^2 = 4\,T_{\mu\,\nu}\,e^{\nu}_{k_1\,\lambda_1}\,P^{\mu\,\alpha}(k_2)\,T_{\alpha\,\beta}\,e^{\beta}_{k_1\,\lambda_1} = -4\,T_{\mu\,\nu}\,T^{\mu}{}_{\beta}\,e^{\nu}_{k_1\,\lambda_1}\,e^{\beta}_{k_1\,\lambda_1} \tag{12.48}$$

Using the compact form of $P^{\mu\,\alpha}(k_2)$, only the first term contributes at this stage. The second term contribution is proportional to
$T_{\mu\,\nu}\,e^{\nu}_{k_1\,\lambda_1}\,k_2^{\mu}\,\bar{k}_2^{\alpha}\,T_{\alpha\,\beta}\,e^{\beta}_{k_1\,\lambda_1}$. Using $k_1 \cdot p_1 = k_2 \cdot p_2$ and $k_1 \cdot p_2 = k_2 \cdot p_1$,

$$k_2^{\mu}\,T_{\mu\,\nu} = (k_2 + p_2 - p_1)_{\nu} = k_{1\,\nu} \tag{12.49}$$

But then $k_{1\,\nu}\,e^{\nu}_{k_1\,\lambda_1} = 0$. Likewise the third term contribution is zero because $k_2^{\alpha}\,T_{\alpha\,\beta} = k_{1\,\beta}$, and $k_{1\,\beta}\,e^{\beta}_{k_1\,\lambda_1} = 0$.

We now sum over λ_1.

$$\sum_{\lambda_1,\,\lambda_2} |M|^2 = -4\,T_{\mu\,\nu}\,P^{\nu\,\beta}(k_1)\,T^{\mu}{}_{\beta} = 4\,\left(T_{\mu\,\nu}\,T^{\mu\,\nu} - 2\right), \tag{12.50}$$

in which all three terms of $P^{\nu\,\beta}(k_1)$ contribute.

For example, the second term is proportional to $\left(T_{\mu\,\nu}\,k_1^{\nu}\,\bar{k}_1^{\beta}\,T^{\mu}{}_{\beta}\right) / \left(k_1 \cdot \bar{k}_1\right)$. The numerator is equal to the denominator, since $k_1^{\nu}\,T_{\mu\,\nu} = (k_1 + p_1 - p_2)_{\mu} = k_{2\,\mu}$, and $k_{2\,\mu}\,T^{\mu}{}_{\beta} = k_{1\,\beta}$, as we saw above.

In the **compton0.mac** code, the expression $T^{\mu\,\nu}$ is represented by **t_munu**, written in terms of the constructs **D**, **UI**, and **Gm** which the Dirac package recognises. The contraction $T_{\mu\,\nu}\,T^{\mu\,\nu}$ becomes **Con (t_munu^2)**.

The batch file **compton0.mac** uses the Dirac package functions **VP** and **ev_Ds**.

```
(%i1) load(work)$
                                  work4 dirac.mac
                                       dgexp
                                       dgcon
                                      dgtrace
                                      dgeval
                                      dgmatrix
massL = [m,M]
indexL = [la,mu,nu,rh,si,ta,al,be,ga]
scalarL = []
" reserved program capital letter name use: "
" Chi, Con, D, Eps, G, G5, Gam, Gm, LI, Nlist, Nlast, UI, P, S, Sig"
"  UU, VP, VV, I2, Z2, CZ2, I4, Z4, CZ4, RZ4, N1,N2,... "

read and interpret file: #pc:/work4/compton0.mac
(%i2) " ======================================================="
(%i3) "  file compton0.mac "
(%i4) "  Maxima by Example, Ch. 12 "
(%i5) "  Dirac Algebra and Quantum Electrodynamics "
(%i6) "  Edwin L Woollett, woollett@charter.net "
(%i7) "  http://www.csulb.edu/~woollett  "
```

```
(%i8) print("      ver: ",_binfo%,"  date: ",mydate)
      ver:  Maxima 5.21.1   date:  2010-09-04

(%i9) " COMPTON SCATTERING OF STRUCTURELESS SPIN 0 PARTICLE "
(%i10) "  which we will call (pi,gamma) scattering "
(%i11) "  gamma(k1,s1) + pi(p1) --> gamma(k2,s2) + pi(p2) "
(%i12) " POPULATE THE LIST invarR OF 4-VEC DOT PRODUCT VALUES. "
(%i13) " Express 4-vector dot products in terms of s, t, u, and m^2 "
(%i14) " Using s = (p1+k1)^2 = (p2+k2)^2, t = (k2-k1)^2 = "
(%i15) " (p2-p1)^2, u = (k2-p1)^2 = (p2-k1)^2, s + t + u = 2*m^2 "
(%i16) invar(D(p1,p1) = m^2,D(p1,k1) = (s-m^2)/2,D(p1,k2) = (-(u-m^2))/2,
             D(p1,p2) = (u+s)/2,D(p2,p2) = m^2,D(p2,k1) = (-(u-m^2))/2,
             D(p2,k2) = (s-m^2)/2,D(k1,k1) = 0,D(k1,k2) = (u+s)/2-m^2,
             D(k2,k2) = 0)
(%i17) " 1. UNPOLARIZED RESULT IN ARBITRARY FRAME (s,t,u form) "
(%i18) " amplitude is M_fi "
(%i19) " M_fi = -2*e^2*( D(e2_cc,e1) - D(e2_cc,p2)*D(e1,p1)/D(k1,p1) "
(%i20) "                + D(e2_cc,p1)*D(e1,p2)/D(k1,p2) ) "
(%i21) " The parenthesis expression can be written as the "
(%i22) " contraction (assume here real pol. vectors ) "
(%i23) "  sum (sum (e2[mu]*T[mu,nu]*e1[mu],mu,0,3),nu,0,3) "
(%i24) " where T[mu,nu] is the expression t_munu : "
(%i25) " -----------------------------------------------"
(%i26) t_munu:UI(p1,mu)*UI(p2,nu)/D(k1,p2)-UI(p1,nu)*UI(p2,mu)/D(k1,p1)
                                          +Gm(mu,nu)
(%i27) "========================================================="
(%i28) "  Note Schwinger's expression (3-12.93) used here, "
(%i29) "  which he derives on page 73, and further discusses p.294"
(%i30) " The Lorentz gauge photon polarization 4-vector always has"
(%i31) " the properties: 1). D(k,e(k,s)) = 0, "
(%i32) "  2). D(n,e(k,s)) = 0, 3). D(kbar,e(k,s)) = 0 "
(%i33) " 4). D(e(k,s)_cc,e(k,r)) = -kroneker-delta(s,r) "
(%i34) " Within the Lorentz gauge one still has gauge freedom "
(%i35) " since one can choose the unit timelike vector n^mu  to"
(%i36) " have any components, with kbar^mu = -k^mu + 2*n^mu*D(k,n). "
(%i37) " In an frame in which n = (1,0,0,0) (which --> D(n,n) = 1)"
(%i38) " the spatial components of kbar are the opposite of those "
(%i39) " of k. One can identify n with any convenient 4-vector in "
(%i40) " the problem "
(%i41) "=========================================================="
(%i42) " Evaluate Mssq = abs. value squared of M_fi."
(%i43) " pull out factor e^4 and sum over polarizations of both photons "
(%i44) " The result is : "
(%i45) Mssq:ratsimp(4*(Con(t_munu^2)-2))
(%o45) ((8*s^2+8*m^4)*u^2-32*m^6*u+8*m^4*s^2-32*m^6*s+40*m^8)
        /((s^2-2*m^2*s+m^4)*u^2+(-2*m^2*s^2+4*m^4*s-2*m^6)*u+m^4*s^2-2*m^6*s
                                  +m^8)
(%i46) " LAB FRAME EVALUATION "
(%i47) " initial pion rest frame evaluation in terms of"
(%i48) " scattering angle of final photon relative to initial"
(%i49) " photon direction (z - x plane) "
(%i50) " kinematics with initial 'pion' p1 at rest. "
(%i51) " initial photon k1 moving along positive z axis. "
(%i52) assume(m > 0,k > 0,kp > 0,th >= 0,th <= %pi)
(%i53) comp_def(p1(m,0,0,0),k1(k,0,0,k),k2(kp,kp*sin(th),0,kp*cos(th)),
                 p2(-kp+k+m,-kp*sin(th),0,k-kp*cos(th)))
(%i54) " -------------------------------------------------------"
(%i55) " conservation of relativistic energy gives "
(%i56) "  m + k = E_p2 + kp "
(%i57) " and conservation of 3 or 4-momentum implies "
(%i58) "  kp = m*k/(m + k*(1-cos(th)))    "
```

```
(%i59) " ---------------------------------------"
(%i60) " find replacement rule for kp using conservation of 4-mom"
(%i61) kp_rule:solve(VP(-k2+k1+p1,-k2+k1+p1) = ev_invar(D(p2,p2)),kp)
(%o61) [kp = -k*m/(k*cos(th)-m-k)]
(%i62) s_th:VP(k1+p1,k1+p1)
(%o62) m^2+2*k*m
(%i63) t_th:(VP(k1-k2,k1-k2),ev(%%,kp_rule))
(%o63) 2*k^2*m/(k*cos(th)-m-k)-2*k^2*m*cos(th)/(k*cos(th)-m-k)
(%i64) u_th:(VP(k2-p1,k2-p1),ev(%%,kp_rule))
(%o64) 2*k*m^2/(k*cos(th)-m-k)+m^2
(%i65) " replace s by s_th, etc "
(%i66) Mssq_lab:factor(sub_stu(Mssq))
(%o66) 4*(cos(th)^2+1)
(%i67) " having pulled out e^4 from Mssq "
(%i68) A:alpha^2*(kp/k)^2/(4*m^2)
(%o68) alpha^2*kp^2/(4*k^2*m^2)
(%i69) " -------------------------------------- "
(%i70) " To get unpolarized differential cross section, "
(%i71) " divide sum over spins Mssq_lab by 2 from average "
(%i72) " over polarization of the  initial photon. "
(%i73) dsigdo_lab_unpol:A*Mssq_lab/2
(%i74) (display2d:true,display(dsigdo_lab_unpol),display2d:false)
                                          2   2     2
                                     alpha  kp  (cos (th) + 1)
                  dsigdo_lab_unpol = -------------------------
                                                  2  2
                                               2 k  m

(%i75) " which agrees with G/R QED p. 437 Eq.(14)  "
(%i76) " since r_0^2 = alpha^2/m^2 "
(%i77) " ======================================="
(%i78) "----------------------------------------------------"
(%i79) " 2. LINEAR PHOTON POLARIZATION CASE     "
(%i80) " ------  lab frame, Coulomb gauge  --------------"
(%i81) " Leave real photon polarization 4-vectors e1,e2 in amplitude. "
(%i82) " replace p1 dot products using lab frame coulomb gauge."
(%i83) " replace p2 dot products with e1,e2 "
(%i84) "                  using 4-mom conservation relation. "
(%i85) " The D(p1,e1) and D(p1,e2) replacements depend on our "
(%i86) "                lab frame Coulomb gauge choices. "
(%i87) " Recall that D has property symmetric, so  "
(%i88) " D(a,b) = D(b,a). "
(%i89) " choose real polarization vectors => linear pol."
(%i90) invar(D(e1,e1) = -1,D(k1,e1) = 0,D(e2,e2) = -1,D(k2,e2) = 0,
             D(p1,e1) = 0,D(p1,e2) = 0,D(e1,p2) = -D(e1,k2),
             D(e2,p2) = D(e2,k1))
(%i91) Ampl:D(e2,c)*D(e1,d)/(u-m^2)+D(e2,a)*D(e1,b)/(s-m^2)-2*D(e1,e2)
(%o91) D(c,e2)*D(d,e1)/(u-m^2)+D(a,e2)*D(b,e1)/(s-m^2)-2*D(e1,e2)
(%i92) " the amplitude is real so just need Ampl^2 for cross section "
(%i93) Ampsq:expand(Ampl^2)
(%o93) D(c,e2)^2*D(d,e1)^2/(u^2-2*m^2*u+m^4)
        +2*D(a,e2)*D(b,e1)*D(c,e2)*D(d,e1)/(s*u-m^2*u-m^2*s+m^4)
        -4*D(c,e2)*D(d,e1)*D(e1,e2)/(u-m^2)
        +D(a,e2)^2*D(b,e1)^2/(s^2-2*m^2*s+m^4)
        -4*D(a,e2)*D(b,e1)*D(e1,e2)/(s-m^2)+4*D(e1,e2)^2
(%i94) "  replace a,b,c,d "
(%i95) Ampsq1:subst([a = k2+2*p2,b = k1+2*p1,c = 2*p1-k2,d = 2*p2-k1],Ampsq)
(%o95) D(e1,2*p2-k1)^2*D(e2,2*p1-k2)^2/(u^2-2*m^2*u+m^4)
        +2*D(e1,2*p1+k1)*D(e1,2*p2-k1)*D(e2,2*p1-k2)*D(e2,2*p2+k2)
         /(s*u-m^2*u-m^2*s+m^4)-4*D(e1,e2)*D(e1,2*p2-k1)*D(e2,2*p1-k2)/(u-m^2)
        +D(e1,2*p1+k1)^2*D(e2,2*p2+k2)^2/(s^2-2*m^2*s+m^4)
        -4*D(e1,e2)*D(e1,2*p1+k1)*D(e2,2*p2+k2)/(s-m^2)+4*D(e1,e2)^2
```

```
(%i96) " expand dot products and use invariants list invarR "
(%i97) Ampsq2:ev_Ds(Ampsq1)
(%o97) 4*D(e1,e2)^2
(%i98) " we see that there is no further reference to s and u in this result"
(%i99) " multiply by A to get polarized cross section "
(%i100) dsigdo_lab_pol:A*Ampsq2
(%i101) (display2d:true,display(dsigdo_lab_pol),display2d:false)
                                              2   2          2
                                         alpha   D (e1, e2) kp
                    dsigdo_lab_pol = ---------------------
                                                  2   2
                                                 k   m

(%i102) " which agrees with G/R QED p. 437, Eq.(12)  "
(%i103) " since r_0^2 = alpha^2/m^2 "
(%i104) " In Coulomb gauge lab frame the four vector dot product square "
(%i105) " D(e2,e1)^2 reduces to dot(e2_vec,e1_vec)^2 "
(%i106) " We recover the unpolarized result above if we sum over both "
(%i107) " e1 and e2 values and divide by 2 since we are averaging over "
(%i108) " the e1 values (see above section on transverse pol. vector sums)"
(%i109) "-------------------------------------------------------------"
(%i110) " If we only average over the polarization of the incident photon"
(%i111) " and leave the polarization of the final photon arbitrary, we "
(%i112) " replace (1+cos(th)^2) by ( 1 - dot(ku,e2_vec)^2) which gives "
(%i113) " 1 for e2_vec chosen perpendicular to the scattering plane, and "
(%i114) " gives sin(th)^2 for e2_vec chosen parallel to the scattering "
(%i115) " plane, leading to the scattered photons polarized preferentially"
(%i116) " in a direction perpendicular to the scattering plane."
```

12.13 compton1.mac: Lepton-photon scattering

The scattering event is

$$e^-(p_1,\sigma_1) + \gamma(k_1,\lambda_1) \to e^-(p_2,\sigma_2) + \gamma(k_2,\lambda_2). \tag{12.51}$$

The invariant amplitude is $M = M_1 + M_2$, where, ignoring the charge squared factor e^2,

$$M_1 = \frac{\bar{u}_2\,\Gamma_1\,u_1}{u - m^2}, \quad \Gamma_1 = \not{e}_1\,(\not{p}_1 - \not{k}_2 + m)\,\not{e}_2^*), \quad u = (p_1 - k_2)^2 \tag{12.52}$$

and

$$M_2 = \frac{\bar{u}_2\,\Gamma_2\,u_1}{s - m^2}, \quad \Gamma_2 = \not{e}_2^*\,(\not{p}_1 + \not{k}_1 + m)\,\not{e}_1, \quad s = (p_1 + k_1)^2 \tag{12.53}$$

The first section of **compton1.mac** calculates the unpolarized differential cross section, proportional to

$$\sum_{\lambda_1,\lambda_2,\sigma_1,\sigma_2} |M_1 + M_2|^2. \tag{12.54}$$

One the the four terms is $\sum |M_1|^2$. If we first sum on λ_1, the result is proportional to

$$\sum_{\lambda_1} \bar{u}_2\,\not{e}_1\,\Gamma_a\,u_1\,\bar{u}_1\,\Gamma_b\,\not{e}_1\,u_2 = \left(\sum_{\lambda_1} e^{\mu}_{k_1\,\lambda_1}\,e^{\tau}_{k_1\,\lambda_1}\right)\,\bar{u}_2\,\gamma_\mu\,\Gamma_a\,u_1\,\bar{u}_1\,\Gamma_b\,\gamma_\tau\,u_2 \tag{12.55}$$

A convenient, simplifying feature of the lepton photon interaction is that only the first (metric tensor) term of $P^{\mu\,\tau}(k_1)$ need be retained, since the other terms produce zero contribution (as is discussed in quantum field theory texts). Thus for

lepton photon interactions we effectively have

$$P^{\mu\nu}(k_1) \Rightarrow -g^{\mu\nu} \tag{12.56}$$

Thus the sum on λ_1 above reduces to

$$-\bar{u}_2\,\gamma_\mu\,\Gamma_a\,u_1\,\bar{u}_1\,\Gamma_b\,\gamma^\mu\,u_2. \tag{12.57}$$

After summing over $\lambda_2,\ \sigma_1,\ \sigma_2$ as well, and again ignoring the factor e^4,

$$\sum_{\lambda_1,\lambda_2,\sigma_1,\sigma_2} |M_1|^2 = \frac{\text{Trace}\ \{(\not{p}_2 + m)\gamma_\mu(\not{p}_1 - \not{k}_2 + m)\gamma_\nu(\not{p}_1 + m)\gamma^\nu(\not{p}_1 - \not{k}_2 + m)\gamma^\mu\}}{(u - m^2)^2} \tag{12.58}$$

The **compton1.mac** code uses the symbol **m1sq** for the trace in the numerator, and uses the symbol **Mssq_unpol** for $\sum |M_1 + M_2|^2$ (see also the run line **(%i32)**). The Dirac package functions **VP**, **D_sub**, **take_parts**,**ts**, and **pullfac** are used in this code.

```
(%i1) load(work)$
                                work4 dirac.mac
                                     dgexp
                                     dgcon
                                    dgtrace
                                    dgeval
                                    dgmatrix
massL = [m,M]
indexL = [la,mu,nu,rh,si,ta,al,be,ga]
scalarL = []
" reserved program capital letter name use: "
" Chi, Con, D, Eps, G, G5, Gam, Gm, LI, Nlist, Nlast, UI, P, S, Sig"
"  UU, VP, VV, I2, Z2, CZ2, I4, Z4, CZ4, RZ4, N1,N2,... "

read and interpret file: #pc:/work4/compton1.mac
(%i2) " ======================================================"
(%i3) "  file compton1.mac "
(%i4) "  Maxima by Example, Ch. 12 "
(%i5) "  Dirac Algebra and Quantum Electrodynamics "
(%i6) "  Edwin L Woollett, woollett@charter.net "
(%i7) "  http://www.csulb.edu/˜woollett  "
(%i8) print("      ver: ",_binfo%,"  date: ",mydate)
      ver:  Maxima 5.21.1   date:  2010-09-06

(%i9) " ---------------------------------------------"
(%i10) "   UNPOLARIZED LEPTON - PHOTON SCATTERING  "
(%i11) " for example : "
(%i12) " ---------------------------------------------"
(%i13) "  e(-,p1,m) + gamma(k1) --> e(-,p2,m) + gamma(k2) "
(%i14) " ----------------------------------------------"
(%i15) "    Arbitrary Frame Covariant Symbolic Methods  "
(%i16) " in an arb frame, D(k1,p1) = D(k2,p2) "
(%i17) "         and D(k1,p2) = D(k2,p1) "
(%i18) " In an arbitrary frame we can express everything "
(%i19) " in terms of m^2, D(k1,p1) and  D(k2,p1) "
(%i20) " with a view to then evaluating in the rest frame"
(%i21) " of the initial electron: p1 = (m,0,0,0) (lab frame) "
(%i22) " ---------------------------------------------"
(%i23) invar(D(p1,p1) = m^2,D(p2,p2) = m^2,D(k1,k1) = 0,D(k2,k2) = 0,
             D(p1,p2) = -D(k2,p1)+D(k1,p1)+m^2,D(k1,p2) = D(k2,p1),
             D(k2,p2) = D(k1,p1),D(k1,k2) = D(k1,p1)-D(k2,p1))
(%i24) m1sq:tr(m+p2,mu,m-k2+p1,nu,m+p1,nu,m-k2+p1,mu)
(%o24) 32*m^4-32*D(k2,p1)*m^2+32*D(k1,p1)*D(k2,p1)
```

```
(%i25) m2sq:tr(m+p2,mu,m+k1+p1,nu,m+p1,nu,m+k1+p1,mu)
(%o25) 32*m^4+32*D(k1,p1)*m^2+32*D(k1,p1)*D(k2,p1)
(%i26) m1m2sq:tr(m+p2,mu,m-k2+p1,nu,m+p1,mu,m+k1+p1,nu)
(%o26) 32*m^4-16*D(k2,p1)*m^2+16*D(k1,p1)*m^2
(%i27) m2m1sq:tr(m+p2,mu,m+k1+p1,nu,m+p1,mu,m-k2+p1,nu)
(%o27) 32*m^4-16*D(k2,p1)*m^2+16*D(k1,p1)*m^2
(%i28) m1m2sq-m2m1sq
(%o28) 0
(%i29) " The interference terms are equal, so  "
(%i30) " ----------------------------------------------"
(%i31) " Form the expression :"
(%i32) "  Mssq_unpol = m1sq/(u-m^2)^2 + m2sq/(s-m^2)^2 "
(%i33) "     + 2*m1m2sq/((u-m^2)*(s-m^2)),  "
(%i34) "     with u-m^2 = -2*D(k2,p1), s-m^2 = 2*D(k1,p1), "
(%i35) "        using u = (p1 - k2)^2, s = (p1 + k1)^2 "
(%i36) " ( we use sloppy notation p^2 for D(p,p) here )"
(%i37) " ----------------------------------------------"
(%i38) Mssq_unpol:expand((-2*m1m2sq)/(4*D(k2,p1)*D(k1,p1))
                          +m2sq/(4*D(k1,p1)^2)+m1sq/(4*D(k2,p1)^2))
(%o38) -16*m^4/(D(k1,p1)*D(k2,p1))+8*m^4/D(k2,p1)^2+8*m^4/D(k1,p1)^2
                                   -16*m^2/D(k2,p1)+16*m^2/D(k1,p1)
                                   +8*D(k2,p1)/D(k1,p1)+8*D(k1,p1)/D(k2,p1)
(%i39) " pull factor of 8 out of definition "
(%i40) Mssq_unpol8:expand(Mssq_unpol/8)
(%o40) -2*m^4/(D(k1,p1)*D(k2,p1))+m^4/D(k2,p1)^2+m^4/D(k1,p1)^2-2*m^2/D(k2,p1)
                                  +2*m^2/D(k1,p1)+D(k2,p1)/D(k1,p1)
                                  +D(k1,p1)/D(k2,p1)
(%i41) "--------------------------------------------------------"
(%i42) "         LAB FRAME EVALUATION UNPOLARIZED               "
(%i43) "          rest frame of initial electron                "
(%i44) " -------------------------------------------------------"
(%i45) assume(m > 0,k > 0,kp > 0,th >= 0,th <= %pi)
(%i46) comp_def(p1(m,0,0,0),k1(k,0,0,k),k2(kp,kp*sin(th),0,kp*cos(th)),
                 p2(-kp+k+m,-kp*sin(th),0,k-kp*cos(th)))
(%i47) " use conservation of 4-momentum to find kp "
(%i48) kp_rule:solve(VP(-k2+k1+p1,-k2+k1+p1) = ev_invar(D(p2,p2)),kp)
(%o48) [kp = -k*m/(k*cos(th)-m-k)]
(%i49) " To compare results, only replace kp in terms 1-5 "
(%i50) " leave kp alone in terms 6-7 "
(%i51) " D_sub (e,[D1,D2,..] uses noncov on each D(p1,p2) in list"
(%i52) M1t5:take_parts(Mssq_unpol8,1,5)
(%o52) -2*m^4/(D(k1,p1)*D(k2,p1))+m^4/D(k2,p1)^2+m^4/D(k1,p1)^2-2*m^2/D(k2,p1)
                                  +2*m^2/D(k1,p1)
(%i53) M6t7:take_parts(Mssq_unpol8,6,7)
(%o53) D(k2,p1)/D(k1,p1)+D(k1,p1)/D(k2,p1)
(%i54) M6t7:D_sub(M6t7,[D(k1,p1),D(k2,p1)])
(%o54) kp/k+k/kp
(%i55) M1t5:D_sub(M1t5,[D(k1,p1),D(k2,p1)])
(%o55) -2*m^2/(k*kp)+m^2/kp^2+m^2/k^2-2*m/kp+2*m/k
(%i56) M1t5:trigsimp(ev(M1t5,kp_rule))
(%o56) cos(th)^2-1
(%i57) " ts is our alternative trigsimp function "
(%i58) M1t5:ts(M1t5,th)
(%o58) -sin(th)^2
(%i59) " restore overall factor of 8 "
(%i60) Mssq_unpol:8*(M6t7+M1t5)
(%o60) 8*(-sin(th)^2+kp/k+k/kp)
(%i61) A:alpha^2*(kp/k)^2/(4*m^2)
(%o61) alpha^2*kp^2/(4*k^2*m^2)
```

```
(%i62) " -------------------------------------- "
(%i63) " To get unpolarized differential cross section, "
(%i64) " divide sum over spins Mssq_unpol by 4 from average "
(%i65) " over spin of initial electron and initial photon. "
(%i66) dsigdo_lab_unpol:A*Mssq_unpol/4
(%i67) (display2d:true,display(dsigdo_lab_unpol),display2d:false)
                                        2   2       2       kp   k
                                   alpha  kp  (- sin (th) + -- + --)
                                                            k    kp
             dsigdo_lab_unpol = ---------------------------------
                                                2  2
                                             2 k  m

(%i68) " which agrees with BLP (86.9) "
```

To calculate the differential cross section for given photon (linear) polarizations (but summing over the inital and final elecron helicities), we retain the photon polarization vector factors $\not{e}_{k_1\lambda_1}$ and $\not{e}_{k_2\lambda_2}$ inside the trace created by summing over the electron helicities.

We already have the basic defining inner products $e_{k_1\lambda_1} \cdot e_{k_1\lambda_1} = -1$, $k_1 \cdot e_{k_1\lambda_1} = 0$, $e_{k_2\lambda_2} \cdot e_{k_2\lambda_2} = -1$, and $k_2 \cdot e_{k_2\lambda_2} = 0$.

Since we want a result appropriate to the rest frame of the initial electron p_1, we identify $n^\mu = p_1^\mu/m$ so that the components of n^μ are $(1, 0, 0, 0)$.

Then $n \cdot e_{k_1\lambda_1} = 0$ implies that $e^0_{k_1\lambda_1} = 0$ and $n \cdot e_{k_2\lambda_2} = 0$ implies that $e^0_{k_2\lambda_2} = 0$.

Then $e_{k_1\lambda_1} = (0, \mathbf{e_{k_1\lambda_1}})$ and $e_{k_2\lambda_2} = (0, \mathbf{e_{k_2\lambda_2}})$.

Since the components of p_1^μ are $(m, 0, 0, 0)$, we then have $p_1 \cdot e_{k_1\lambda_1} = 0$ and $p_1 \cdot e_{k_2\lambda_2} = 0$. And using 4-momentum conservation, the former equation implies $0 = e_{k_1\lambda_1} \cdot (k_2 + p_2 - k_1)$, and hence $e_{k_1\lambda_1} \cdot p_2 = -e_{k_1\lambda_1} \cdot k_2$.

Finally we replace $e_{k_2\lambda_2} \cdot p_2 = e_{k_2\lambda_2} \cdot k_1$, using again 4-momentum conservation and the previous relations.

In our code, **e1** stands for $e_{k_1\lambda_1}$ and **e2** stands for $e_{k_2\lambda_2}$.

```
(%i69) "======================================================"
(%i70) "----UNPOLARIZED ELECTRONS, POLARIZED PHOTONS ------------"
(%i71) "------- rest frame of initial electron p1 ------------- "
(%i72) " Leave photon polarization 4-vectors e1,e2 in trace. "
(%i73) " replace p1 dot products using p1 rest frame choice."
(%i74) " replace p2 dot products with e1,e2 "
(%i75) "                 using 4-mom conservation relation. "
(%i76) " The D(p1,e1) and D(p1,e2) replacements depend on our "
(%i77) "               p1 rest frame choice."
(%i78) " Recall that D has property symmetric, so  "
(%i79) " D(a,b) = D(b,a). "
(%i80) invar(D(e1,e1) = -1,D(k1,e1) = 0,D(e2,e2) = -1,D(k2,e2) = 0,
             D(p1,e1) = 0,D(p1,e2) = 0,D(e1,p2) = -D(e1,k2),
             D(e2,p2) = D(e2,k1))
(%i81) M1sq:tr(m+p2,e1,m-k2+p1,e2,m+p1,e2,m-k2+p1,e1)
(%o81) 8*D(k1,p1)*D(k2,p1)-16*D(e1,k2)^2*D(k2,p1)
(%i82) M2sq:tr(m+p2,e2,m+k1+p1,e1,m+p1,e1,m+k1+p1,e2)
(%o82) 8*D(k1,p1)*D(k2,p1)+16*D(e2,k1)^2*D(k1,p1)
```

```
(%i83) M1m2sq:tr(m+p2,e1,m-k2+p1,e2,m+p1,e1,m+k1+p1,e2)
(%o83) -16*D(e1,e2)^2*D(k1,p1)*D(k2,p1)+8*D(k1,p1)*D(k2,p1)
                                       +8*D(e2,k1)^2*D(k2,p1)
                                       -8*D(e1,k2)^2*D(k1,p1)
(%i84) M2m1sq:tr(m+p2,e2,m+k1+p1,e1,m+p1,e2,m-k2+p1,e1)
(%o84) -16*D(e1,e2)^2*D(k1,p1)*D(k2,p1)+8*D(k1,p1)*D(k2,p1)
                                       +8*D(e2,k1)^2*D(k2,p1)
                                       -8*D(e1,k2)^2*D(k1,p1)
(%i85) M2m1sq-M1m2sq
(%o85) 0
(%i86) " The interference terms are equal, so  "
(%i87) Mssq_pol:expand((-2*M1m2sq)/(4*D(k2,p1)*D(k1,p1))
                     +M2sq/(4*D(k1,p1)^2)+M1sq/(4*D(k2,p1)^2))
(%o87) 2*D(k2,p1)/D(k1,p1)+2*D(k1,p1)/D(k2,p1)+8*D(e1,e2)^2-4
(%i88) " avoid use of noncov(D(e1,e2)) which would expand D(e1,e2)"
(%i89) " in terms of undefined components, by instead using ev"
(%i90) " or D_sub  for selected replacements. "
(%i91) Mssq_pol:D_sub(Mssq_pol,[D(k1,p1),D(k2,p1)])
(%o91) 2*kp/k+2*k/kp+8*D(e1,e2)^2-4
(%i92) Mssq_pol:pullfac(Mssq_pol,2)
(%o92) 2*(kp/k+k/kp+4*D(e1,e2)^2-2)
(%i93) "To get electron unpolarized, photon polarized differential cross section,"
(%i94) " divide  Mssq_pol by 2 from average over helicity of initial electron. "
(%i95) dsigdo_lab_pol:A*Mssq_pol/2
(%i96) (display2d:true,display(dsigdo_lab_pol),display2d:false)
                                   2   2  kp   k       2
                          alpha  kp  (-- + -- + 4 D (e1, e2) - 2)
                                      k    kp
          dsigdo_lab_pol = ---------------------------------------
                                                  2  2
                                               4 k  m

(%i97) " which agrees with the Klein-Nishina form quoted by Weinberg (8.7.39)"
(%i98) "=========================================================="
(%i99) " Now assume unpolarized initial photon. Average  the "
(%i100) " polarized photon dsigdo(r,s) over the two possible values "
(%i101) " of r which specifies the transverse polarization "
(%i102) " vector of incident photon e1_vec. With n^mu = (1,0,0,0)"
(%i103) " D(e1,e2)^2 --> { dot3 (e1_vec,e2_vec) }^2 "
(%i104) " and sum (D(e1(r),e2(s))^2,r,1,2) --> 1 - {dot3(ku,e2_vec(s)}^2"
(%i105) " with ku = unit 3-vector along the z axis, dot3(a,b) = 3-vec dot prod. "
(%i106) " and dsigdo(s) = sum(dsigdo(r,s),r,1,2)/2 "
(%i107) " The overall result is recovered with: "
(%i108) dsigdo_s:subst(4*D(e1,e2)^2 = 2-2*dot3(ku,e2_vec)^2,dsigdo_lab_pol)
(%i109) (display2d:true,display(dsigdo_s),display2d:false)
                              2   2          2               kp   k
                     alpha  kp  (- 2 dot3 (ku, e2_vec) + -- + --)
                                                             k    kp
           dsigdo_s = --------------------------------------------
                                               2  2
                                            4 k  m

(%i110) "------------------------------------------------------"
(%i111) "  which agrees with Weinberg (8.7.40) "
(%i112) " Summing the above over the two possible values of "
(%i113) " final photon polarization s then leads to our "
(%i114) " unpolarized result above "
```

12.14 pair1.mac: Unpolarized Two Photon Pair Annihilation

The pair annihilation to two photon event is

$$e^-(p_1, \sigma_1) + e^+(p_2, \sigma_2) \to \gamma(k_1, \lambda_1) + \gamma(k_2, \lambda_2). \tag{12.59}$$

With the definitions

$$s = (p_1 + p_2)^2, \quad t = (p_1 - k_1)^2, \quad u = (p_1 - k_2)^2, \tag{12.60}$$

and the abbreviations

$$u_1 = u(p_1, \sigma_1), \quad \bar{v}_2 = \bar{v}(p_2, \sigma_2), \quad e_1^* = e_{k_1,\lambda_1}^*, \quad e_2^* = e_{k_2,\lambda_2}^*, \tag{12.61}$$

$M = M_1 + M_2$, with

$$M_1 = \frac{e^2\, \bar{v}_2\, \not{e}_2^*\, (\not{p}_1 - \not{k}_1 + m)\, \not{e}_1^*\, u_1}{t - m^2} \tag{12.62}$$

and

$$M_2 = \frac{e^2\, \bar{v}_2\, \not{e}_1^*\, (\not{p}_1 - \not{k}_2 + m)\, \not{e}_2^*\, u_1}{u - m^2}. \tag{12.63}$$

Some references for this calculation are Griffith p. 241, G/R QED: (p. 192 rest frame of electron, p. 198 CM frame), P/S p. 168, BLP p. 370, Schwinger p. 314, Kaku p. 170, I/Z p. 230, and B/D p. 132.

The batch file **pair1.mac** works out the unpolarized differential cross section first in an arbitrary frame, and then in the CM frame. To compare our result with G/R in the CM frame, we express the result in terms of **(E,v)**, where **p = E*v, E^2 = m^2 + p^2**, with **v** (the dimensionless velocity of either of the incident leptons relative to the CM frame) having values $0 \leq v < 1$.

The Dirac package functions **D_sub** and **ts** are used.

```
(%i1) load(work)$
                                work4 dirac.mac
                                     dgexp
                                     dgcon
                                    dgtrace
                                    dgeval
                                    dgmatrix
massL = [m,M]
indexL = [la,mu,nu,rh,si,ta,al,be,ga]
scalarL = []
" reserved program capital letter name use: "
" Chi, Con, D, Eps, G, G5, Gam, Gm, LI, Nlist, Nlast, UI, P, S, Sig"
"  UU, VP, VV, I2, Z2, CZ2, I4, Z4, CZ4, RZ4, N1,N2,... "

read and interpret file: #pc:/work4/pair1.mac
(%i2) " ======================================================"
(%i3) "  file pair1.mac "
(%i4) "  Maxima by Example, Ch. 12 "
(%i5) "  Dirac Algebra and Quantum Electrodynamics "
(%i6) "  Edwin L Woollett, woollett@charter.net "
(%i7) "  http://www.csulb.edu/~woollett  "
(%i8) print("      ver: ",_binfo%," date: ",mydate)
      ver:  Maxima 5.21.1   date:  2010-09-08
```

```
(%i9) " ------------------------------------------------"
(%i10) "      UNPOLARIZED PAIR ANNIHILATION          "
(%i11) " ------------------------------------------------"
(%i12) "  e(-,p1) + e(+,p2) --> gamma(k1) + gamma(k2) "
(%i13) " ------------------------------------------------"
(%i14) "   Arbitrary Frame Covariant Symbolic Methods  "
(%i15) " ------------------------------------------------"
(%i16) " POPULATE THE LIST invarR OF 4-VEC DOT PRODUCT VALUES. "
(%i17) " Express 4-vector dot products in terms of t, u, and m^2 "
(%i18) " Using s = (p1+p2)^2 = (k1+k2)^2, t = (p1-k1)^2 = "
(%i19) " (k2-p2)^2, u = (p1-k2)^2 = (k1-p2)^2, s + t + u = 2*m^2 "
(%i20) " ------------------------------------------------------"
(%i21) invar(D(p1,p1) = m^2,D(p1,k1) = (m^2-t)/2,D(p1,k2) = (m^2-u)/2,
             D(p1,p2) = (-(u+t))/2,D(p2,p2) = m^2,D(p2,k1) = (m^2-u)/2,
             D(p2,k2) = (m^2-t)/2,D(k1,k1) = 0,D(k1,k2) = m^2-(u+t)/2,
             D(k2,k2) = 0)
(%i22) "-------------------------------------------------------"
(%i23) "  sum |M|^2 over spins of electrons and polarizations "
(%i24) "    of the  photons, which leads to the expression: "
(%i25) " ---------------------------------------------------"
(%i26) "      m1sq/(t-m^2)^2 + m2sq/(u-m^2)^2 +           "
(%i27) "           (m1m2sq+m2m1sq)/((t-m^2)*(u-m^2))"
(%i28) " ---------------------------------------------------"
(%i29) " --    where (ignoring e^4)  ---                    "
(%i30) m1sq:tr(p2-m,mu,m-k1+p1,nu,m+p1,nu,m-k1+p1,mu)
(%o30) 8*t*u-8*m^2*u-24*m^2*t-8*m^4
(%i31) m2sq:tr(p2-m,mu,m-k2+p1,nu,m+p1,nu,m-k2+p1,mu)
(%o31) 8*t*u-24*m^2*u-8*m^2*t-8*m^4
(%i32) m1m2sq:tr(p2-m,mu,m-k1+p1,nu,m+p1,mu,m-k2+p1,nu)
(%o32) -8*m^2*u-8*m^2*t-16*m^4
(%i33) m2m1sq:tr(p2-m,mu,m-k2+p1,nu,m+p1,mu,m-k1+p1,nu)
(%o33) -8*m^2*u-8*m^2*t-16*m^4
(%i34) m1m2sq-m2m1sq
(%o34) 0
(%i35) Mssq:ratsimp(2*m1m2sq/((t-m^2)*(u-m^2))+m2sq/(u-m^2)^2+m1sq/(t-m^2)^2)
(%o35) ((8*t-8*m^2)*u^3+(24*m^4-56*m^2*t)*u^2+(8*t^3-56*m^2*t^2+112*m^4*t)*u
                -8*m^2*t^3+24*m^4*t^2-48*m^8)
        /((t^2-2*m^2*t+m^4)*u^2+(-2*m^2*t^2+4*m^4*t-2*m^6)*u+m^4*t^2-2*m^6*t
                        +m^8)
(%i36) "----- replace t and u in terms of dot products in arb frame  ------ "
(%i37) Mssq_dot:expand(subst([t = m^2-2*D(k1,p1),u = m^2-2*D(k2,p1)],Mssq))
(%o37) -16*m^4/(D(k1,p1)*D(k2,p1))-8*m^4/D(k2,p1)^2-8*m^4/D(k1,p1)^2
                          +16*m^2/D(k2,p1)+16*m^2/D(k1,p1)
                          +8*D(k2,p1)/D(k1,p1)+8*D(k1,p1)/D(k2,p1)
(%i38) "===================================================="
(%i39) "  specialize to center of momentum frame  "
(%i40) "----------------------------------------------------"
(%i41) " kinematics in center of momentum frame "
(%i42) " p_vec_mag : E*v, v = velocity rel to CM frame "
(%i43) " E^2 = m^2 + p^2 = m^2/(1 - v^2) "
(%i44) " initial electron 3-mom in dir. of pos z axis"
(%i45) " e(-,p1) e(+,p2) --> gamma(k1) gamma(k2) "
(%i46) " CM energy = 2*E shared equally by incident"
(%i47) " fermions and outgoing photons "
(%i48) assume(m > 0,E > 0,th >= 0,th <= %pi)
(%i49) comp_def(p1(E,0,0,v*E),p2(E,0,0,-v*E),k1(E,E*sin(th),0,E*cos(th)),
         k2(E,-E*sin(th),0,-E*cos(th)))
(%i50) " pf/pi = E/p = E/(vE) = 1/v "
(%i51) " dsigdo_CM = A*|M|^2 for given pol, where "
(%i52) " having absorbed e^4 into A, e^2 = 4*%pi*alpha"
(%i53) A:alpha^2/(16*v*E^2)
(%o53) alpha^2/(16*v*E^2)
```

```
(%i54) " convert to a function of angle th between "
(%i55) " p1_vec and k1_vec "
(%i56) Mssq_th:D_sub(Mssq_dot,[D(k1,p1),D(k2,p1)])
(%o56) -16*m^4/((E^2-cos(th)*v*E^2)*(cos(th)*v*E^2+E^2))
        +8*(E^2-cos(th)*v*E^2)/(cos(th)*v*E^2+E^2)+16*m^2/(cos(th)*v*E^2+E^2)
        -8*m^4/(cos(th)*v*E^2+E^2)^2+16*m^2/(E^2-cos(th)*v*E^2)
        -8*m^4/(E^2-cos(th)*v*E^2)^2+8*(cos(th)*v*E^2+E^2)/(E^2-cos(th)*v*E^2)
(%i57) Mssq_th:trigsimp(subst(E^2 = m^2/(1-v^2),Mssq_th))
(%o57) -((16*sin(th)^4+16)*v^4-32*sin(th)^2*v^2-16)
        /(cos(th)^4*v^4-2*cos(th)^2*v^2+1)
(%i58) " simplify the denominator - override default factor "
(%i59) mt1d:denom(Mssq_th)
(%o59) cos(th)^4*v^4-2*cos(th)^2*v^2+1
(%i60) mt1d:ratsubst(x,v^2*cos(th)^2,mt1d)
(%o60) x^2-2*x+1
(%i61) mt1d:factor(mt1d)
(%o61) (x-1)^2
(%i62) mt1d:subst(x = v^2*cos(th)^2,mt1d)
(%o62) (cos(th)^2*v^2-1)^2
(%i63) " simplify the numerator "
(%i64) mt1n:num(Mssq_th)
(%o64) -(16*sin(th)^4+16)*v^4+32*sin(th)^2*v^2+16
(%i65) " pull out factor of 16 from numerator "
(%i66) mt1n16:factor(mt1n)/16
(%o66) -sin(th)^4*v^4-v^4+2*sin(th)^2*v^2+1
(%i67) " change sin(th) to cos(th) in numerator "
(%i68) mt1n16:expand(ts(mt1n16,th))
(%o68) -cos(th)^4*v^4+2*cos(th)^2*v^4-2*v^4-2*cos(th)^2*v^2+2*v^2+1
(%i69) " simplified num/denom expression: "
(%i70) Mssq_unpol:16*mt1n16/mt1d
(%o70) 16*(-cos(th)^4*v^4+2*cos(th)^2*v^4-2*v^4-2*cos(th)^2*v^2+2*v^2+1)
        /(cos(th)^2*v^2-1)^2
(%i71) " divide by 4 from averaging over initial spin "
(%i72) " and polarization "
(%i73) dsigdo_CM:A*Mssq_unpol/4
(%i74) (display2d:true,display(dsigdo_CM),display2d:false)
dsigdo_CM =
          2        4      4        2      4      4         2      2      2
       alpha  (- cos  (th) v  + 2 cos (th) v  - 2 v  - 2 cos (th) v  + 2 v  + 1)
       ----------------------------------------------------------------------
                                     2      2     2  2
                             4 v (cos  (th) v  - 1)  E

(%i75) " which agrees with Greiner and Reinhardt, p.201, eqn (17)"
(%i76) "----------------------------------------------------------"
```

12.15 pair2.mac: Polarized Two Photon Pair Annihilation Amplitudes

The pair annihilation to two photon event is

$$e^-(p_1, \sigma_1) + e^+(p_2, \sigma_2) \to \gamma(k_1, \lambda_1) + \gamma(k_2, \lambda_2). \tag{12.64}$$

With the definitions

$$s = (p_1 + p_2)^2, \quad t = (p_1 - k_1)^2, \quad u = (p_1 - k_2)^2. \tag{12.65}$$

If we use explicit Dirac spinors and matrices and photon polarization 3-vectors to calculate polarized amplitudes in which both the leptons and photons have definite helicities in the CM frame, we start with the polarized amplitude already written down in the previous section dealing with **pair1.mac**.

With the abbreviations

$$u_1 = u(p_1, \sigma_1), \quad \bar{v}_2 = \bar{v}(p_2, \sigma_2), \quad e_1^* = e_{k_1,\lambda_1}^*, \quad e_2^* = e_{k_2,\lambda_2}^*, \tag{12.66}$$

$M = M_1 + M_2$, with

$$M_1 = \frac{e^2\, \bar{v}_2\, \not{e}_2^*\, (\not{p}_1 - \not{k}_1 + m)\, \not{e}_1^*\, u_1}{t - m^2} \tag{12.67}$$

and

$$M_2 = \frac{e^2\, \bar{v}_2\, \not{e}_1^*\, (\not{p}_1 - \not{k}_2 + m)\, \not{e}_2^*\, u_1}{u - m^2}. \tag{12.68}$$

Here we use explicit Dirac spinor methods in the CM frame, letting each lepton have the energy **`E = M/2`**, with **`M`** the total CM frame energy, and then each lepton has 3-momentum magnitude **`p = E*v = M*v/2`**. We also have the dimensionless lepton CM frame speed **`v = sqrt( 1 - 4*m^2/M^2 )`**. Each outgoing photon has energy **`k = M/2`** and each photon has 3-momentum magnitude **`k`**.

We use the photon polarization conventions of F. Gross. (See Franz Gross, Rel. Q. Mech. and Fld. Theory, pages 54, 218, and Schwinger, PSF I, Sec. 3-13-98.)

The outgoing photons are chosen to be travelling along the $+z$ and $-z$ axes. We let $\mathbf{e}(\hat{z}, \lambda)$ be the photon polarization 3-vector describing a photon travelling in the $+z$ direction, with $\lambda = 1$ being a positive helicity state and $\lambda = -1$ being a negative helicity state. Then

$$\mathbf{e}(\hat{z}, 1) = -\frac{1}{\sqrt{2}}\,(\hat{x} + i\,\hat{y}), \tag{12.69}$$

and

$$\mathbf{e}(\hat{z}, -1) = \frac{1}{\sqrt{2}}\,(\hat{x} - i\,\hat{y}). \tag{12.70}$$

We let $\mathbf{e}(-\hat{z}, \lambda)$ be the photon polarization 3-vector describing a photon travelling in the $-z$ direction, with $\lambda = 1$ being a positive helicity state and $\lambda = -1$ being a negative helicity state. Then

$$\mathbf{e}(-\hat{z}, 1) = \frac{1}{\sqrt{2}}\,(\hat{x} - i\,\hat{y}) = R_y(\pi)\,\mathbf{e}(\hat{z}, 1), \tag{12.71}$$

and

$$\mathbf{e}(-\hat{z}, -1) = -\frac{1}{\sqrt{2}}\,(\hat{x} + i\,\hat{y}) = R_y(\pi)\,\mathbf{e}(\hat{z}, -1). \tag{12.72}$$

For our calculation, since the photons are in the final state, we need complex conjugates of these. The first example discussed is **`RR --> RR`**, which means two right-handed electrons (actually positive helicity) turn into two positive helicity photons. For this first example, we define CM frame components with the line:

```
comp_def(p1(M/2,M*v*sin(th)/2,0,M*v*cos(th)/2),k1(M/2,0,0,M/2),
            k2(M/2,0,0,(-M)/2),e1_cc(0,(-1)/sqrt(2),%i/sqrt(2),0),
            e2_cc(0,1/sqrt(2),%i/sqrt(2),0))
```

which shows that the photon with 4-momentum **`k1`** is travelling along the positive **`z`** axis, and the polarization state of this photon is described by the 4-vector **`e1_cc`** with the components **`(0,(-1)/sqrt(2),%i/sqrt(2),0)`** in which the three spatial components agree with $\mathbf{e}(\hat{z}, 1)^*$.

Likewise, the photon with 4-momentum **k2** is travelling along the negative **z** axis, and the symbol **e2_cc** is the 4-vector describing its positive helicity state, with components **(0,1/sqrt(2),%i/sqrt(2),0)** whose spatial components agree with $\mathbf{e}(-\hat{z}, 1)^*$.

The cases considered are **RR --> RR**, **RR --> LL**, **RR --> RL**, **RR --> LR**, **RL --> RL**, **RL --> LR**, and **RL --> RR** (this last case has amplitude equal to zero).

Since we want to compare our results with Schwinger's results, some involved manipulations are necessary in each case to arrive at a comparable form.

```
(%i1) load(work)$
                                work4 dirac.mac

                                     dgexp

                                     dgcon

                                    dgtrace

                                    dgeval

                                    dgmatrix

massL = [m,M]

indexL = [la,mu,nu,rh,si,ta,al,be,ga]

scalarL = []

" reserved program capital letter name use: "

" Chi, Con, D, Eps, G, G5, Gam, Gm, LI, Nlist, Nlast, UI, P, S, Sig"

"  UU, VP, VV, I2, Z2, CZ2, I4, Z4, CZ4, RZ4, N1,N2,... "


read and interpret file: #pc:/work4/pair2.mac
(%i2) " ======================================================"
(%i3) "  file pair2.mac "
(%i4) "  Maxima by Example, Ch. 12 "
(%i5) "  Dirac Algebra and Quantum Electrodynamics "
(%i6) "  Edwin L Woollett, woollett@charter.net "
(%i7) "  http://www.csulb.edu/~woollett  "
(%i8) print("      ver: ",_binfo%,"  date: ",mydate)
      ver:  Maxima 5.21.1   date:  2010-09-08

(%i9) " --------------------------------------------"
(%i10) "   POLARIZED PAIR ANNIHILATION in CM Frame    "
(%i11) " --------------------------------------------"
(%i12) "  e(-,p1) + e(+,p2) --> gamma(k1) + gamma(k2)  "
(%i13) " ---------------------------------------------"
(%i14) "   AMPLITUDES USING EXPLICIT DIRAC SPINORS    "
(%i15) " ---------------------------------------------"
(%i16) " Define the needed Dirac spinors and barred spinors. "
(%i17) " For example, RR --> RR amplitude without the factor e^2, and"
(%i18) " either t-m^2 or u-m^2 in the denominator . The photon 3-momenta"
(%i19) " magnitudes are equal to M/2, where M = 2*E = CM frame energy."
(%i20) " M >= 2*m, and 'low energy' corresponds to M at its lower limit,"
(%i21) " for which v is close to value 0, and 'high energy' corresponds"
(%i22) " to M >> m, for which v approaches its maximum value 1 "
```

```
(%i23) " The lepton 3-momentum magnitudes are each equal to E*v = M*v/2 "
(%i24) " where v is the dimensionless velocity with "
(%i25) " v^2 = p^2/E^2 = (E^2-m^2)/E^2 = 1- 4*m^2/M^2 "
(%i26) "  All particle energies are equal to M/2."
(%i27) " ----------------------------------------------------"
(%i28) " CASE: e(-,p1,+1)  e(+,p2,+1) --> gamma(k1,+1)  gamma(k2,+1) "
(%i29) " ----------------------------------------------------"
(%i30) " Both the electron and positron have helicity +1. "
(%i31) " Each of the final gammas have helicity +1. "
(%i32) "For a given set of helicity quantum numbers,  the amplitude is the"
(%i33) " sum of two terms:  Mtn/(t-m^2) + Mun/(u-m^2) , where  "
(%i34) " Mtd = t-m^2 = -2*D(k1,p1), Mud = u-m^2 = -2*D(k2,p1). "
(%i35) " --------------------------------------------------"
(%i36) " To simplify the photon helicity 3-vectors, let k1_3vec be "
(%i37) " along the positive z axis, and k2_3vec be along the "
(%i38) " minus z axis, p1_3vec in the z-x plane at an angle of "
(%i39) " th radians to k1_3vec."
(%i40) (up1:UU(M/2,M*v/2,th,0,1),vp2b:sbar(VV(M/2,M*v/2,%pi-th,%pi,1)))
(%i41) " ------------------------------------------------------"
(%i42) " We need to define the components of the four vectors "
(%i43) " p1, k1, k2, e1_cc, and e2_cc relative to the CM frame"
(%i44) " For the photon polarization 4-vectors, we use the phase"
(%i45) " conventions of Franz Gross, Rel. Q. Mech. and Fld. Theory"
(%i46) " pages 54, 218, and 230. e1_cc means complex conjugate of e1. "
(%i47) " Without the 'cc', e1 means here e (z_hat,+1) "
(%i48) " and e2 means here e (-z_hat,+1) "
(%i49) comp_def(p1(M/2,M*v*sin(th)/2,0,M*v*cos(th)/2),k1(M/2,0,0,M/2),
                k2(M/2,0,0,(-M)/2),e1_cc(0,(-1)/sqrt(2),%i/sqrt(2),0),
                e2_cc(0,1/sqrt(2),%i/sqrt(2),0))
(%i50) " denominators "
(%i51) Mtd:-2*pullfac(noncov(D(k1,p1)),M^2/4)
(%o51) -(1-cos(th)*v)*M^2/2
(%i52) Mud:-2*pullfac(noncov(D(k2,p1)),M^2/4)
(%o52) -(cos(th)*v+1)*M^2/2
(%i53) g1:sL(e2_cc) . (m*I4-sL(k1)+sL(p1)) . sL(e1_cc)
(%o53) matrix([2*m,0,-2*(M/2-cos(th)*v*M/2),0],[0,0,0,0],
              [-2*(cos(th)*v*M/2-M/2),0,2*m,0],[0,0,0,0])
(%i54) g2:sL(e1_cc) . (m*I4-sL(k2)+sL(p1)) . sL(e2_cc)
(%o54) matrix([0,0,0,0],[0,2*m,0,-2*(cos(th)*v*M/2+M/2)],[0,0,0,0],
              [0,-2*(-cos(th)*v*M/2-M/2),0,2*m])
(%i55) " M1 = Mtn/Mtd "
(%i56) Mtn:vp2b . g1 . up1
(%o56) sin((%pi-th)/2)*sqrt((v+1)*M/2)
                      *(2*m*cos(th/2)*sqrt((v+1)*M/2)
                       -2*cos(th/2)*sqrt(-(v-1)*M/2)*(cos(th)*v*M/2-M/2))
        -sin((%pi-th)/2)*sqrt(-(v-1)*M/2)
                        *(2*m*cos(th/2)*sqrt(-(v-1)*M/2)
                         -2*cos(th/2)*sqrt((v+1)*M/2)*(M/2-cos(th)*v*M/2))
(%i57) expand(ratsubst(cos(th/2),sin((%pi-th)/2),%))
(%o57) -cos(th/2)^2*cos(th)*sqrt(1-v)*v*sqrt(v+1)*M^2
        +cos(th/2)^2*sqrt(1-v)*sqrt(v+1)*M^2+2*m*cos(th/2)^2*v*M
(%i58) expand(ratsubst(2*m/M,sqrt(1-v^2),rootscontract(%)))
(%o58) -2*m*cos(th/2)^2*cos(th)*v*M+2*m*cos(th/2)^2*v*M+2*m*cos(th/2)^2*M
(%i59) pullfac(%,2*m*M*cos(th/2)^2)
(%o59) 2*m*cos(th/2)^2*(-cos(th)*v+v+1)*M
(%i60) expand(ratsubst((cos(th)+1)/2,cos(th/2)^2,%))
(%o60) -m*cos(th)^2*v*M+m*v*M+m*cos(th)*M+m*M
(%i61) Mtn:pullfac(%,m*M)
(%o61) m*(-cos(th)^2*v+v+cos(th)+1)*M
(%i62) M1:Mtn/Mtd
(%o62) -2*m*(-cos(th)^2*v+v+cos(th)+1)/((1-cos(th)*v)*M)
```

```
(%i63) " M2 = Mun/Mud "
(%i64) Mun:vp2b . g2 . up1
(%o64) cos((%pi-th)/2)*sqrt((v+1)*M/2)
                    *(2*m*sin(th/2)*sqrt((v+1)*M/2)
                     -2*sin(th/2)*sqrt(-(v-1)*M/2)*(-cos(th)*v*M/2-M/2))
        -cos((%pi-th)/2)*sqrt(-(v-1)*M/2)
                      *(2*m*sin(th/2)*sqrt(-(v-1)*M/2)
                       -2*sin(th/2)*sqrt((v+1)*M/2)*(cos(th)*v*M/2+M/2))
(%i65) expand(ratsubst(sin(th/2),cos((%pi-th)/2),%))
(%o65) sin(th/2)^2*cos(th)*sqrt(1-v)*v*sqrt(v+1)*M^2
        +sin(th/2)^2*sqrt(1-v)*sqrt(v+1)*M^2+2*m*sin(th/2)^2*v*M
(%i66) expand(ratsubst(2*m/M,sqrt(1-v^2),rootscontract(%)))
(%o66) 2*m*sin(th/2)^2*cos(th)*v*M+2*m*sin(th/2)^2*v*M+2*m*sin(th/2)^2*M
(%i67) pullfac(%,2*m*M*sin(th/2)^2)
(%o67) 2*m*sin(th/2)^2*(cos(th)*v+v+1)*M
(%i68) Mun:expand(ratsubst((1-cos(th))/2,sin(th/2)^2,%))
(%o68) -m*cos(th)^2*v*M+m*v*M-m*cos(th)*M+m*M
(%i69) Mun:pullfac(%,m*M)
(%o69) m*(-cos(th)^2*v+v-cos(th)+1)*M
(%i70) M2:Mun/Mud
(%o70) -2*m*(-cos(th)^2*v+v-cos(th)+1)/((cos(th)*v+1)*M)
(%i71) " Add amplitudes and simplify trig functions "
(%i72) Mfi:trigsimp(M2+M1)
(%o72) (4*m*v+4*m)/((cos(th)^2*v^2-1)*M)
(%i73) " factor numerator and pull out minus sign in denom "
(%i74) Mfi_n:factor(num(Mfi))
(%o74) 4*m*(v+1)
(%i75) Mfi_d:pullfac(expand(denom(Mfi)),-M)
(%o75) -(1-cos(th)^2*v^2)*M
(%i76) Mfi:Mfi_n/Mfi_d
(%o76) -4*m*(v+1)/((1-cos(th)^2*v^2)*M)
(%i77) (display2d:true,
        disp("CASE: e(-,p1,+1)  e(+,p2,+1) --> gamma(k1,+1)  gamma(k2,+1) "),
        display(Mfi),display2d:false)
         CASE: e(-,p1,+1)  e(+,p2,+1) --> gamma(k1,+1)  gamma(k2,+1)

                                        4 m (v + 1)
                       Mfi = - -------------------
                                            2      2
                                (1 - cos (th) v ) M

(%i78) " which agrees with Schwinger, PSF I, (3-13-98)"
(%i79) "---------------------------------------------"
(%i80) "CASE: e(-,p1,+1)  e(+,p2,+1) --> gamma(k1,-1)  gamma(k2,-1) "
(%i81) " ------------------------------------------------"
(%i82) comp_def(e1_cc(0,1/sqrt(2),%i/sqrt(2),0),
                 e2_cc(0,(-1)/sqrt(2),%i/sqrt(2),0))
(%i83) listarray(e1_cc)
(%o83) [0,1/sqrt(2),%i/sqrt(2),0]
(%i84) listarray(e2_cc)
(%o84) [0,-1/sqrt(2),%i/sqrt(2),0]
(%i85) " the spinors and denominators are the same "
(%i86) " recompute the g1 and g2 matrices "
(%i87) g1:sL(e2_cc) . (m*I4-sL(k1)+sL(p1)) . sL(e1_cc)
(%o87) matrix([0,0,0,0],[0,2*m,0,-2*(cos(th)*v*M/2-M/2)],[0,0,0,0],
              [0,-2*(M/2-cos(th)*v*M/2),0,2*m])
(%i88) g2:sL(e1_cc) . (m*I4-sL(k2)+sL(p1)) . sL(e2_cc)
(%o88) matrix([2*m,0,-2*(-cos(th)*v*M/2-M/2),0],[0,0,0,0],
              [-2*(cos(th)*v*M/2+M/2),0,2*m,0],[0,0,0,0])
(%i89) " -------------------------------------"
```

```
(%i90) " M1 = Mtn/Mtd "
(%i91) Mtn:vp2b . g1 . up1
(%o91) cos((%pi-th)/2)*sqrt((v+1)*M/2)
                      *(2*m*sin(th/2)*sqrt((v+1)*M/2)
                       -2*sin(th/2)*sqrt(-(v-1)*M/2)*(M/2-cos(th)*v*M/2))
        -cos((%pi-th)/2)*sqrt(-(v-1)*M/2)
                        *(2*m*sin(th/2)*sqrt(-(v-1)*M/2)
                         -2*sin(th/2)*sqrt((v+1)*M/2)*(cos(th)*v*M/2-M/2))
(%i92) expand(ratsubst(sin(th/2),cos((%pi-th)/2),%))
(%o92) sin(th/2)^2*cos(th)*sqrt(1-v)*v*sqrt(v+1)*M^2
        -sin(th/2)^2*sqrt(1-v)*sqrt(v+1)*M^2+2*m*sin(th/2)^2*v*M
(%i93) expand(ratsubst(2*m/M,sqrt(1-v^2),rootscontract(%)))
(%o93) 2*m*sin(th/2)^2*cos(th)*v*M+2*m*sin(th/2)^2*v*M-2*m*sin(th/2)^2*M
(%i94) pullfac(%,2*m*M*sin(th/2)^2)
(%o94) 2*m*sin(th/2)^2*(cos(th)*v+v-1)*M
(%i95) expand(ratsubst((1-cos(th))/2,sin(th/2)^2,%))
(%o95) -m*cos(th)^2*v*M+m*v*M+m*cos(th)*M-m*M
(%i96) Mtn:pullfac(%,m*M)
(%o96) m*(-cos(th)^2*v+v+cos(th)-1)*M
(%i97) M1:Mtn/Mtd
(%o97) -2*m*(-cos(th)^2*v+v+cos(th)-1)/((1-cos(th)*v)*M)
(%i98) "-----------------------------"
(%i99) " M2 = Mun/Mud "
(%i100) Mun:vp2b . g2 . up1
(%o100) sin((%pi-th)/2)*sqrt((v+1)*M/2)
                      *(2*m*cos(th/2)*sqrt((v+1)*M/2)
                       -2*cos(th/2)*sqrt(-(v-1)*M/2)*(cos(th)*v*M/2+M/2))
         -sin((%pi-th)/2)*sqrt(-(v-1)*M/2)
                        *(2*m*cos(th/2)*sqrt(-(v-1)*M/2)
                         -2*cos(th/2)*sqrt((v+1)*M/2)*(-cos(th)*v*M/2-M/2))
(%i101) expand(ratsubst(cos(th/2),sin((%pi-th)/2),%))
(%o101) -cos(th/2)^2*cos(th)*sqrt(1-v)*v*sqrt(v+1)*M^2
         -cos(th/2)^2*sqrt(1-v)*sqrt(v+1)*M^2+2*m*cos(th/2)^2*v*M
(%i102) expand(ratsubst(2*m/M,sqrt(1-v^2),rootscontract(%)))
(%o102) -2*m*cos(th/2)^2*cos(th)*v*M+2*m*cos(th/2)^2*v*M-2*m*cos(th/2)^2*M
(%i103) pullfac(%,2*m*M*cos(th/2)^2)
(%o103) 2*m*cos(th/2)^2*(-cos(th)*v+v-1)*M
(%i104) expand(ratsubst((cos(th)+1)/2,cos(th/2)^2,%))
(%o104) -m*cos(th)^2*v*M+m*v*M-m*cos(th)*M-m*M
(%i105) Mun:pullfac(%,m*M)
(%o105) m*(-cos(th)^2*v+v-cos(th)-1)*M
(%i106) M2:Mun/Mud
(%o106) -2*m*(-cos(th)^2*v+v-cos(th)-1)/((cos(th)*v+1)*M)
(%i107) " Add amplitudes and simplify trig functions "
(%i108) Mfi:trigsimp(M2+M1)
(%o108) (4*m*v-4*m)/((cos(th)^2*v^2-1)*M)
(%i109) " factor numerator and pull out minus sign in denom "
(%i110) Mfi_n:pullfac(num(Mfi),-4*m)
(%o110) -4*m*(1-v)
(%i111) Mfi_d:pullfac(expand(denom(Mfi)),-M)
(%o111) -(1-cos(th)^2*v^2)*M
(%i112) Mfi:Mfi_n/Mfi_d
(%o112) 4*m*(1-v)/((1-cos(th)^2*v^2)*M)
```

```
(%i113) (display2d:true,
         disp("CASE: e(-,p1,+1)  e(+,p2,+1) --> gamma(k1,-1)  gamma(k2,-1) "),
         display(Mfi),display2d:false)
         CASE: e(-,p1,+1)  e(+,p2,+1) --> gamma(k1,-1)  gamma(k2,-1)

                                     4 m (1 - v)
                           Mfi = -------------------
                                            2      2
                                  (1 - cos (th) v ) M

(%i114) " which agrees with Schwinger, PSF I, (3-13-98)"
(%i115) "-------------------------------------------------"
(%i116) "CASE: e(-,p1,+1)  e(+,p2,+1) --> gamma(k1,1)  gamma(k2,-1) "
(%i117) " -----------------------------------------------------"
(%i118) comp_def(e1_cc(0,(-1)/sqrt(2),%i/sqrt(2),0))
(%i119) listarray(e1_cc)
(%o119) [0,-1/sqrt(2),%i/sqrt(2),0]
(%i120) " the spinors and denominators are the same "
(%i121) " recompute the g1 and g2 matrices "
(%i122) g1:sL(e2_cc) . (m*I4-sL(k1)+sL(p1)) . sL(e1_cc)
(%o122) matrix([0,0,0,0],[0,0,sin(th)*v*M,0],[0,0,0,0],[-sin(th)*v*M,0,0,0])
(%i123) g2:sL(e1_cc) . (m*I4-sL(k2)+sL(p1)) . sL(e2_cc)
(%o123) matrix([0,0,0,0],[0,0,sin(th)*v*M,0],[0,0,0,0],[-sin(th)*v*M,0,0,0])
(%i124) " -------------------------------------"
(%i125) " M1 = Mtn/Mtd "
(%i126) Mtn:vp2b . g1 . up1
(%o126) -2*cos((%pi-th)/2)*cos(th/2)*sin(th)*v*M*sqrt(-(v-1)*M/2)
          *sqrt((v+1)*M/2)
(%i127) " M1 = Mtn/Mtd "
(%i128) Mtn:vp2b . g1 . up1
(%o128) -2*cos((%pi-th)/2)*cos(th/2)*sin(th)*v*M*sqrt(-(v-1)*M/2)
          *sqrt((v+1)*M/2)
(%i129) expand(ratsubst(sin(th/2),cos((%pi-th)/2),%))
(%o129) -cos(th/2)*sin(th/2)*sin(th)*sqrt(1-v)*v*sqrt(v+1)*M^2
(%i130) expand(ratsubst(2*m/M,sqrt(1-v^2),rootscontract(%)))
(%o130) -2*m*cos(th/2)*sin(th/2)*sin(th)*v*M
(%i131) Mtn:ratsubst(sin(th)/2,cos(th/2)*sin(th/2),%)
(%o131) -m*sin(th)^2*v*M
(%i132) M1:Mtn/Mtd
(%o132) 2*m*sin(th)^2*v/((1-cos(th)*v)*M)
(%i133) "--------------------------------"
(%i134) " M2 = Mun/Mud "
(%i135) Mun:vp2b . g2 . up1
(%o135) -2*cos((%pi-th)/2)*cos(th/2)*sin(th)*v*M*sqrt(-(v-1)*M/2)
          *sqrt((v+1)*M/2)
(%i136) expand(ratsubst(sin(th/2),cos((%pi-th)/2),%))
(%o136) -cos(th/2)*sin(th/2)*sin(th)*sqrt(1-v)*v*sqrt(v+1)*M^2
(%i137) expand(ratsubst(2*m/M,sqrt(1-v^2),rootscontract(%)))
(%o137) -2*m*cos(th/2)*sin(th/2)*sin(th)*v*M
(%i138) Mun:ratsubst(sin(th)/2,cos(th/2)*sin(th/2),%)
(%o138) -m*sin(th)^2*v*M
(%i139) M2:Mun/Mud
(%o139) 2*m*sin(th)^2*v/((cos(th)*v+1)*M)
(%i140) " ------------------------------------------"
(%i141) " Add amplitudes and simplify trig functions "
(%i142) Mfi:trigsimp(M2+M1)
(%o142) -4*m*sin(th)^2*v/((cos(th)^2*v^2-1)*M)
(%i143) Mfi:num(Mfi)/pullfac(expand(denom(Mfi)),-M)
(%o143) 4*m*sin(th)^2*v/((1-cos(th)^2*v^2)*M)
```

```
(%i144) (display2d:true,
         disp("CASE: e(-,p1,+1)  e(+,p2,+1) --> gamma(k1,+1)  gamma(k2,-1) "),
         display(Mfi),display2d:false)
         CASE: e(-,p1,+1)  e(+,p2,+1) --> gamma(k1,+1)  gamma(k2,-1)

                                              2
                                       4 m sin  (th) v
                              Mfi = -------------------
                                                2      2
                                      (1 - cos  (th) v ) M

(%i145) " ---------------------------------------------------"
(%i146) "-----------------------------------------------"
(%i147) "CASE: e(-,p1,+1)  e(+,p2,+1) --> gamma(k1,-1)  gamma(k2,+1) "
(%i148) " ---------------------------------------------------"
(%i149) comp_def(e1_cc(0,1/sqrt(2),%i/sqrt(2),0),
                 e2_cc(0,1/sqrt(2),%i/sqrt(2),0))
(%i150) listarray(e1_cc)
(%o150) [0,1/sqrt(2),%i/sqrt(2),0]
(%i151) listarray(e2_cc)
(%o151) [0,1/sqrt(2),%i/sqrt(2),0]
(%i152) " the spinors and denominators are the same "
(%i153) " recompute the g1 and g2 matrices "
(%i154) g1:sL(e2_cc) . (m*I4-sL(k1)+sL(p1)) . sL(e1_cc)
(%o154) matrix([0,0,0,sin(th)*v*M],[0,0,0,0],[0,-sin(th)*v*M,0,0],[0,0,0,0])
(%i155) g2:sL(e1_cc) . (m*I4-sL(k2)+sL(p1)) . sL(e2_cc)
(%o155) matrix([0,0,0,sin(th)*v*M],[0,0,0,0],[0,-sin(th)*v*M,0,0],[0,0,0,0])
(%i156) " ---------------------------------------"
(%i157) " ---------------------------------------"
(%i158) " M1 = Mtn/Mtd "
(%i159) Mtn:vp2b . g1 . up1
(%o159) -2*sin((%pi-th)/2)*sin(th/2)*sin(th)*v*M*sqrt(-(v-1)*M/2)
          *sqrt((v+1)*M/2)
(%i160) expand(ratsubst(cos(th/2),sin((%pi-th)/2),%))
(%o160) -cos(th/2)*sin(th/2)*sin(th)*sqrt(1-v)*v*sqrt(v+1)*M^2
(%i161) expand(ratsubst(2*m/M,sqrt(1-v^2),rootscontract(%)))
(%o161) -2*m*cos(th/2)*sin(th/2)*sin(th)*v*M
(%i162) Mtn:ratsubst(sin(th)/2,cos(th/2)*sin(th/2),%)
(%o162) -m*sin(th)^2*v*M
(%i163) M1:Mtn/Mtd
(%o163) 2*m*sin(th)^2*v/((1-cos(th)*v)*M)
(%i164) "-------------------------------"
(%i165) " M2 = Mun/Mud "
(%i166) Mun:vp2b . g2 . up1
(%o166) -2*sin((%pi-th)/2)*sin(th/2)*sin(th)*v*M*sqrt(-(v-1)*M/2)
          *sqrt((v+1)*M/2)
(%i167) expand(ratsubst(cos(th/2),sin((%pi-th)/2),%))
(%o167) -cos(th/2)*sin(th/2)*sin(th)*sqrt(1-v)*v*sqrt(v+1)*M^2
(%i168) expand(ratsubst(2*m/M,sqrt(1-v^2),rootscontract(%)))
(%o168) -2*m*cos(th/2)*sin(th/2)*sin(th)*v*M
(%i169) Mun:ratsubst(sin(th)/2,cos(th/2)*sin(th/2),%)
(%o169) -m*sin(th)^2*v*M
(%i170) M2:Mun/Mud
(%o170) 2*m*sin(th)^2*v/((cos(th)*v+1)*M)
(%i171) " ----------------------------------"
(%i172) Mfi:trigsimp(M2+M1)
(%o172) -4*m*sin(th)^2*v/((cos(th)^2*v^2-1)*M)
(%i173) Mfi:num(Mfi)/pullfac(expand(denom(Mfi)),-M)
(%o173) 4*m*sin(th)^2*v/((1-cos(th)^2*v^2)*M)
```

```
(%i174) (display2d:true,
         disp("CASE: e(-,p1,+1)  e(+,p2,+1) --> gamma(k1,-1)  gamma(k2,+1) "),
         display(Mfi),display2d:false)
         CASE: e(-,p1,+1)  e(+,p2,+1) --> gamma(k1,-1)  gamma(k2,+1)

                                              2
                                      4 m sin (th) v
                             Mfi = -------------------
                                             2      2
                                     (1 - cos (th) v ) M

(%i175) " -------------------------------------------------"
(%i176) " Schwinger's amplitude for these last two cases differs by a minus sign "
(%i177) " ------------------------------------------------ "
(%i178) " CASES LEPTONS HAVE OPPOSITE HELICITIES: see Schwinger (3-13-97). "
(%i179) "-----------------------------------------------"
(%i180) "CASE: e(-,p1,+1)  e(+,p2,-1) --> gamma(k1,+1)  gamma(k2,-1) "
(%i181) " --------------------------------------------------"
(%i182) " recompute vp2b "
(%i183) vp2b:sbar(VV(M/2,M*v/2,%pi-th,%pi,-1))
(%i184) " denominators Mtd, Mud remain the same "
(%i185) " recompute g1 and g2 "
(%i186) comp_def(e1_cc(0,(-1)/sqrt(2),%i/sqrt(2),0),
                 e2_cc(0,(-1)/sqrt(2),%i/sqrt(2),0))
(%i187) listarray(e1_cc)
(%o187) [0,-1/sqrt(2),%i/sqrt(2),0]
(%i188) listarray(e2_cc)
(%o188) [0,-1/sqrt(2),%i/sqrt(2),0]
(%i189) g1:sL(e2_cc) . (m*I4-sL(k1)+sL(p1)) . sL(e1_cc)
(%o189) matrix([0,0,0,0],[0,0,sin(th)*v*M,0],[0,0,0,0],[-sin(th)*v*M,0,0,0])
(%i190) g2:sL(e1_cc) . (m*I4-sL(k2)+sL(p1)) . sL(e2_cc)
(%o190) matrix([0,0,0,0],[0,0,sin(th)*v*M,0],[0,0,0,0],[-sin(th)*v*M,0,0,0])
(%i191) " ------------------------------------"
(%i192) " ------------------------------------"
(%i193) " M1 = Mtn/Mtd "
(%i194) Mtn:vp2b . g1 . up1
(%o194) sin((%pi-th)/2)*cos(th/2)*sin(th)*v*(v+1)*M^2/2
         -sin((%pi-th)/2)*cos(th/2)*sin(th)*(v-1)*v*M^2/2
(%i195) Mtn:expand(ratsubst(cos(th/2),sin((%pi-th)/2),%))
(%o195) cos(th/2)^2*sin(th)*v*M^2
(%i196) M1:Mtn/Mtd
(%o196) -2*cos(th/2)^2*sin(th)*v/(1-cos(th)*v)
(%i197) "-------------------------------"
(%i198) " M2 = Mun/Mud "
(%i199) Mun:vp2b . g2 . up1
(%o199) sin((%pi-th)/2)*cos(th/2)*sin(th)*v*(v+1)*M^2/2
         -sin((%pi-th)/2)*cos(th/2)*sin(th)*(v-1)*v*M^2/2
(%i200) Mun:expand(ratsubst(cos(th/2),sin((%pi-th)/2),%))
(%o200) cos(th/2)^2*sin(th)*v*M^2
(%i201) M2:Mun/Mud
(%o201) -2*cos(th/2)^2*sin(th)*v/(cos(th)*v+1)
(%i202) " ------------------------------"
(%i203) trigsimp(M2+M1)
(%o203) 4*cos(th/2)^2*sin(th)*v/(cos(th)^2*v^2-1)
(%i204) Mfi:(-num(%))/(-denom(%))
(%o204) -4*cos(th/2)^2*sin(th)*v/(1-cos(th)^2*v^2)
```

```
(%i205) (display2d:true,
         disp("CASE: e(-,p1,+1)  e(+,p2,-1) --> gamma(k1,+1)  gamma(k2,-1) "),
         display(Mfi),display2d:false)
         CASE: e(-,p1,+1)  e(+,p2,-1) --> gamma(k1,+1)  gamma(k2,-1)

                                          2 th
                                     4 cos  (--) sin(th) v
                                              2
                             Mfi = - --------------------
                                                2      2
                                       1 - cos (th) v

(%i206) " -------------------------------------------------"
(%i207) " This differs by a sign compared to Schwinger. "
(%i208) " -----------------------------------------------"
(%i209) "CASE: e(-,p1,+1)  e(+,p2,-1) --> gamma(k1,-1)  gamma(k2,+1) "
(%i210) " -------------------------------------------------"
(%i211) " spinors and denominators are the same , recompute g1 and g2"
(%i212) comp_def(e1_cc(0,1/sqrt(2),%i/sqrt(2),0),
                 e2_cc(0,1/sqrt(2),%i/sqrt(2),0))
(%i213) listarray(e1_cc)
(%o213) [0,1/sqrt(2),%i/sqrt(2),0]
(%i214) listarray(e2_cc)
(%o214) [0,1/sqrt(2),%i/sqrt(2),0]
(%i215) g1:sL(e2_cc) . (m*I4-sL(k1)+sL(p1)) . sL(e1_cc)
(%o215) matrix([0,0,0,sin(th)*v*M],[0,0,0,0],[0,-sin(th)*v*M,0,0],[0,0,0,0])
(%i216) g2:sL(e1_cc) . (m*I4-sL(k2)+sL(p1)) . sL(e2_cc)
(%o216) matrix([0,0,0,sin(th)*v*M],[0,0,0,0],[0,-sin(th)*v*M,0,0],[0,0,0,0])
(%i217) " M1 = Mtn/Mtd "
(%i218) Mtn:vp2b . g1 . up1
(%o218) cos((%pi-th)/2)*sin(th/2)*sin(th)*(v-1)*v*M^2/2
         -cos((%pi-th)/2)*sin(th/2)*sin(th)*v*(v+1)*M^2/2
(%i219) Mtn:expand(ratsubst(sin(th/2),cos((%pi-th)/2),%))
(%o219) -sin(th/2)^2*sin(th)*v*M^2
(%i220) M1:Mtn/Mtd
(%o220) 2*sin(th/2)^2*sin(th)*v/(1-cos(th)*v)
(%i221) Mun:vp2b . g2 . up1
(%o221) cos((%pi-th)/2)*sin(th/2)*sin(th)*(v-1)*v*M^2/2
         -cos((%pi-th)/2)*sin(th/2)*sin(th)*v*(v+1)*M^2/2
(%i222) Mun:expand(ratsubst(sin(th/2),cos((%pi-th)/2),%))
(%o222) -sin(th/2)^2*sin(th)*v*M^2
(%i223) M2:Mun/Mud
(%o223) 2*sin(th/2)^2*sin(th)*v/(cos(th)*v+1)
(%i224) " -----------------------------"
(%i225) trigsimp(M2+M1)
(%o225) -4*sin(th/2)^2*sin(th)*v/(cos(th)^2*v^2-1)
(%i226) Mfi:(-num(%))/(-denom(%))
(%o226) 4*sin(th/2)^2*sin(th)*v/(1-cos(th)^2*v^2)
(%i227) (display2d:true,
         disp("CASE: e(-,p1,+1)  e(+,p2,-1) --> gamma(k1,-1)  gamma(k2,+1) "),
         display(Mfi),display2d:false)
         CASE: e(-,p1,+1)  e(+,p2,-1) --> gamma(k1,-1)  gamma(k2,+1)

                                         2 th
                                    4 sin  (--) sin(th) v
                                             2
                              Mfi = --------------------
                                               2      2
                                      1 - cos (th) v

(%i228) " -------------------------------------------------"
(%i229) " This differs by a sign compared to Schwinger. "
```

```
(%i230) " ------------------------------------------------"
(%i231) "CASE: e(-,p1,+1)  e(+,p2,-1) --> gamma(k1,+1)  gamma(k2,+1) "
(%i232) " ------------------------------------------------"
(%i233) comp_def(e1_cc(0,(-1)/sqrt(2),%i/sqrt(2),0),
                 e2_cc(0,1/sqrt(2),%i/sqrt(2),0))
(%i234) listarray(e1_cc)
(%o234) [0,-1/sqrt(2),%i/sqrt(2),0]
(%i235) listarray(e2_cc)
(%o235) [0,1/sqrt(2),%i/sqrt(2),0]
(%i236) g1:sL(e2_cc) . (m*I4-sL(k1)+sL(p1)) . sL(e1_cc)
(%o236) matrix([2*m,0,-2*(M/2-cos(th)*v*M/2),0],[0,0,0,0],
               [-2*(cos(th)*v*M/2-M/2),0,2*m,0],[0,0,0,0])
(%i237) g2:sL(e1_cc) . (m*I4-sL(k2)+sL(p1)) . sL(e2_cc)
(%o237) matrix([0,0,0,0],[0,2*m,0,-2*(cos(th)*v*M/2+M/2)],[0,0,0,0],
               [0,-2*(-cos(th)*v*M/2-M/2),0,2*m])
(%i238) " M1 = Mtn/Mtd "
(%i239) Mtn:vp2b . g1 . up1
(%o239) cos((%pi-th)/2)*sqrt(-(v-1)*M/2)
                       *(2*m*cos(th/2)*sqrt((v+1)*M/2)
                        -2*cos(th/2)*sqrt(-(v-1)*M/2)*(cos(th)*v*M/2-M/2))
         -cos((%pi-th)/2)*sqrt((v+1)*M/2)
                         *(2*m*cos(th/2)*sqrt(-(v-1)*M/2)
                          -2*cos(th/2)*sqrt((v+1)*M/2)*(M/2-cos(th)*v*M/2))
(%i240) expand(ratsubst(sin(th/2),cos((%pi-th)/2),%))
(%o240) cos(th/2)*sin(th/2)*M^2-cos(th/2)*sin(th/2)*cos(th)*v*M^2
(%i241) expand(ratsubst(sin(th)/2,cos(th/2)*sin(th/2),%))
(%o241) sin(th)*M^2/2-cos(th)*sin(th)*v*M^2/2
(%i242) Mtn:pullfac(%,sin(th)*M^2/2)
(%o242) sin(th)*(1-cos(th)*v)*M^2/2
(%i243) M1:Mtn/Mtd
(%o243) -sin(th)
(%i244) Mun:vp2b . g2 . up1
(%o244) sin((%pi-th)/2)*sqrt((v+1)*M/2)
                       *(2*m*sin(th/2)*sqrt(-(v-1)*M/2)
                        -2*sin(th/2)*sqrt((v+1)*M/2)*(cos(th)*v*M/2+M/2))
         -sin((%pi-th)/2)*sqrt(-(v-1)*M/2)
                         *(2*m*sin(th/2)*sqrt((v+1)*M/2)
                          -2*sin(th/2)*sqrt(-(v-1)*M/2)*(-cos(th)*v*M/2-M/2))
(%i245) expand(ratsubst(cos(th/2),sin((%pi-th)/2),%))
(%o245) -cos(th/2)*sin(th/2)*cos(th)*v*M^2-cos(th/2)*sin(th/2)*M^2
(%i246) expand(ratsubst(sin(th)/2,cos(th/2)*sin(th/2),%))
(%o246) -cos(th)*sin(th)*v*M^2/2-sin(th)*M^2/2
(%i247) Mun:factor(%)
(%o247) -sin(th)*(cos(th)*v+1)*M^2/2
(%i248) M2:Mun/Mud
(%o248) sin(th)
(%i249) Mfi:M2+M1
(%o249) 0
(%i250) (display2d:true,
         disp("CASE: e(-,p1,+1)  e(+,p2,-1) --> gamma(k1,+1)  gamma(k2,+1) "),
         display(Mfi),display2d:false)
         CASE: e(-,p1,+1)  e(+,p2,-1) --> gamma(k1,+1)  gamma(k2,+1)

                                    Mfi = 0

(%i251) " The amplitude is zero for this case. "
```

12.16 moller3.mac: Squared Polarized Amplitudes Using Symbolic or Explicit Matrix Trace Methods

We provid examples of using trace methods for the square of a polarized amplitude in the case of arbitrary energy (finite mass) Moller scattering. This detailed verification using both explicit matrix traces and the symbolic traces takes several minutes to carry out for each case, with the symbolic **nc_tr** method (equivalent to **noncov (tr(..))**) requiring about five times as much computation time as the explicit matrix method **m_tr**.

The new feature is the **tr** or **m_tr** argument **S(sv, Sp)**, in which either **sv = 1** or **sv = -1**, and **Sp** is a symbol for the particle spin 4-vector associated with the particle's 4-momentum.

In the explicit matrix method, such an argument inserts a matrix factor **P(sv,Sp)**, which turns into the matrix **(I4 + sv*Gam[5].sL(Sp))/2**. This is the matrix version of the finite mass spin projection operator, and requires a frame dependent definition (using **comp_def**) of the components of the spin 4-vector corresponding to a particle's 4-momentum. (See our section on high energy physics notation.)

When used with the symbolic **tr** method, the result is a factor $\frac{1}{2}(1 + \sigma\,\gamma^5\,\not{S})$, with $\sigma = \pm 1$ being the value of **sv** and $\not{S}$ being $(Sp)_\mu\,\gamma^\mu$.

Here is the batch file **moller3.mac**:

```
/* file moller3.mac
   m_tr and nc_tr comparisons of
   the square of polarized amplitudes
   compared with dirac spinor results
   found in moller2.mac
   */

  " moller3.mac "$
    "*********************************************"$
   print ("      ver: ",_binfo%,"  date: ",mydate )$
  " Maxima by Example, Ch. 12 "$
  " Dirac Algebra and Quantum Electrodynamics "$
  " Edwin L. Woollett "$
  " http://www.csulb.edu/~woollett  "$
  " woollett@charter.net "$
  "SQUARED POLARIZED AMPLITUDES VIA SYMBOLIC AND MATRIX TRACE METHODS"$
  "  FOR ARBITRARY ENERGY MOLLER SCATTERING  "$
  "  e(-,p1,sv1) + e(-,p2,sv2) --> e(-,p3,sv3) + e(-,p4,sv4) "$
  " ------------------------------------------------------"$

  invar (D(p1,p1) = m^2,
         D(p2,p2) = m^2,
         D(p3,p3) = m^2,
         D(p4,p4) = m^2,
         D(p1,Sp1) = 0,
         D(Sp1,Sp1) = -1,
         D(p2,Sp2) = 0,
         D(Sp2,Sp2) = -1,
         D(p3,Sp3) = 0,
         D(Sp3,Sp3) = -1,
         D(p4,Sp4) = 0,
         D(Sp4,Sp4) = -1 )$
```

```
 comp_def ( p1( E,0,0,p),
            Sp1 (p/m,0,0,E/m),
            p2( E,0,0,-p),
            Sp2 (p/m,0,0,-E/m),
            p3 (E,p*sin(th),0,p*cos(th)),
            Sp3 (p/m,E*sin(th)/m,0,E*cos(th)/m),
            p4 (E,-p*sin(th),0,-p*cos(th)),
            Sp4 (p/m,-E*sin(th)/m,0,-E*cos(th)/m))$



p_Em (expr) := expand (ratsubst(E^2-m^2,p^2,expr))$

E_pm (expr) := expand (ratsubst (p^2 + m^2,E^2,expr))$


s_th : VP (p1+p2,p1+p2);

t_th : VP (p1-p3,p1-p3);

u_th : VP (p1-p4,p1-p4);


t_th2 : to_ao2 (t_th,th);

u_th2 : to_ao2 (u_th,th);

t_thE : p_Em(t_th);

u_thE : p_Em(u_th);

 load("MSQcomp.mac")$
```

and here is the automated comparison code which Moller3.mac loads:

```
/* file MSQcomp.mac */

/* comparison  code for moller3.mac */
  disp (" MSQcomp() to compare matrix trace, symbolic trace
  methods for square of polarized amplitudes
  with square of spinor amplitudes. This is an
  automated comparison, assuming A_spinor is
  globally defined and also global sv1,sv2,sv3,sv4.
  To see progress details, set details to true.")$
     details:false$
```

```
 /*************************************************************/
MSQcomp () :=

   block(

   if not details then (disp (" this comparison could take 3 plus minutes ")),

       A_spinor_th : fr_ao2 (A_spinor,th),

        A_spSQ : E_pm (A_spinor_th^2),

       if details then display (A_spSQ),


   M1n_m : trigsimp (E_pm (mcon ( m_tr (S(sv3,Sp3),p3+m,mu,S(sv1,Sp1),p1+m,nu)*
                 m_tr (S(sv4,Sp4),p4+m,mu,S(sv2,Sp2),p2+m,nu), mu,nu))),

     if details then (disp ("M1n_m"), display (M1n_m)),


  M1n_s : trigsimp (E_pm (econ ( nc_tr (S(sv3,Sp3),p3+m,mu,S(sv1,Sp1),p1+m,nu)*
                 nc_tr (S(sv4,Sp4),p4+m,mu,S(sv2,Sp2),p2+m,nu), mu,nu))),

     if details then (disp ("M1n_s"),display (M1n_s)),


    M1n_diff : M1n_s - M1n_m,

    if details then print ("  M1n_s - M1n_m = ",M1n_diff),

    if M1n_diff # 0 then (
            disp ("M1n symbolic  method # matrix method disagree "),
            return (M1n_diff)),



   M2n_m : trigsimp (E_pm (mcon ( m_tr (S(sv4,Sp4),p4+m,mu,S(sv1,Sp1),p1+m,nu)*
                m_tr (S(sv3,Sp3),p3+m,mu,S(sv2,Sp2),p2+m,nu), mu,nu))),

    if details then disp ("M2n_m"),


  M2n_s : trigsimp (E_pm (econ ( nc_tr (S(sv4,Sp4),p4+m,mu,S(sv1,Sp1),p1+m,nu)*
                nc_tr (S(sv3,Sp3),p3+m,mu,S(sv2,Sp2),p2+m,nu), mu,nu))),

    if details then disp ("M2n_s"),

    M2n_diff : M2n_s - M2n_m,
```

```
    if details then print ("  M2n_s - M2n_m = ",M2n_diff),

    if M2n_diff # 0 then (
            disp ("M2n symbolic  method # matrix method disagree "),
            return (M2n_diff)),

    M12n_m : trigsimp (E_pm (mcon ( m_tr (S(sv3,Sp3),p3+m,mu,S(sv1,Sp1),p1+m,
                         nu,S(sv4,Sp4),p4+m,mu,S(sv2,Sp2),p2+m,nu),mu,nu))),

    if details then disp ("M12n_m"),

    M12n_s : trigsimp (E_pm (nc_tr (S(sv3,Sp3),p3+m,mu,S(sv1,Sp1),p1+m,
                         nu,S(sv4,Sp4),p4+m,mu,S(sv2,Sp2),p2+m,nu))),

    if details then disp ("M12n_s"),


    M12n_diff : M12n_s - M12n_m,

    if details then print ("  M12n_s - M12n_m = ",M12n_diff),

    if M12n_diff # 0 then (
            disp ("M21n symbolic  method # matrix method disagree "),
            return (M21n_diff)),



    M21n_m : trigsimp (E_pm (mcon ( m_tr (S(sv4,Sp4),p4+m,mu,S(sv1,Sp1),p1+m,
                      nu,S(sv3,Sp3),p3+m,mu,S(sv2,Sp2),p2+m,nu),mu,nu))),

    if details then disp ("M21n_m"),


    M21n_s : trigsimp (E_pm (nc_tr (S(sv4,Sp4),p4+m,mu,S(sv1,Sp1),p1+m,
                         nu,S(sv3,Sp3),p3+m,mu,S(sv2,Sp2),p2+m,nu))),

    if details then disp ("M21n_s"),


    if details then print ("  M21n_s - M21n_m = ",M21n_s - M21n_m),

    M21n_diff : M21n_s - M21n_m,

    if details then print ("  M21n_s - M21n_m = ",M21n_diff),

    if M21n_diff # 0 then (
            disp ("M21n symbolic  method # matrix method disagree "),
            return (M21n_diff)),
```

```
MfiSQ : expand (M1n_m/t_th^2 + M2n_m/u_th^2 - M12n_m/(t_th*u_th)
             - M21n_m/(t_th*u_th)),

MfiSQ : trigsimp (MfiSQ),


MSQ_diff : trigsimp (MfiSQ - A_spSQ),

if MSQ_diff = 0 then disp (" agreement ")

 else  disp (" no agreement "),

 display ([sv1,sv2,sv3,sv4]))$
```

and here are some cases worked out:

```
(%i1) load(work)$
                                work4 dirac.mac
                                     dgexp
                                     dgcon
                                    dgtrace
                                    dgeval
                                    dgmatrix
massL = [m,M]
indexL = [la,mu,nu,rh,si,ta,al,be,ga]
scalarL = []
" reserved program capital letter name use: "
" Chi, Con, D, Eps, G, G5, Gam, Gm, LI, Nlist, Nlast, UI, P, S, Sig"
"  UU, VP, VV, I2, Z2, CZ2, I4, Z4, CZ4, RZ4, N1,N2,... "

read and interpret file: #pc:/work4/moller3.mac
(%i2) " moller3.mac "
(%i3) "*********************************************"
(%i4) print("      ver: ",_binfo%," date: ",mydate)
      ver:  Maxima 5.21.1   date:  2010-09-11

(%i5) " Maxima by Example, Ch. 12 "
(%i6) " Dirac Algebra and Quantum Electrodynamics "
(%i7) " Edwin L. Woollett "
(%i8) " http://www.csulb.edu/~woollett  "
(%i9) " woollett@charter.net "
(%i10) "SQUARED POLARIZED AMPLITUDES VIA SYMBOLIC AND MATRIX TRACE METHODS"
(%i11) "  FOR ARBITRARY ENERGY MOLLER SCATTERING  "
(%i12) "  e(-,p1,sv1) + e(-,p2,sv2) --> e(-,p3,sv3) + e(-,p4,sv4) "
(%i13) " -------------------------------------------------------"
(%i14) invar(D(p1,p1) = m^2,D(p2,p2) = m^2,D(p3,p3) = m^2,D(p4,p4) = m^2,
             D(p1,Sp1) = 0,D(Sp1,Sp1) = -1,D(p2,Sp2) = 0,D(Sp2,Sp2) = -1,
             D(p3,Sp3) = 0,D(Sp3,Sp3) = -1,D(p4,Sp4) = 0,D(Sp4,Sp4) = -1)
(%i15) comp_def(p1(E,0,0,p),Sp1(p/m,0,0,E/m),p2(E,0,0,-p),Sp2(p/m,0,0,(-E)/m),
                p3(E,p*sin(th),0,p*cos(th)),
                Sp3(p/m,E*sin(th)/m,0,E*cos(th)/m),
                p4(E,-p*sin(th),0,-p*cos(th)),
                Sp4(p/m,(-E*sin(th))/m,0,(-E*cos(th))/m))
(%i16) p_Em(expr):=expand(ratsubst(E^2-m^2,p^2,expr))
(%i17) E_pm(expr):=expand(ratsubst(m^2+p^2,E^2,expr))
```

```
(%i18) s_th:VP(p2+p1,p2+p1)
(%o18) 4*E^2
(%i19) t_th:VP(p1-p3,p1-p3)
(%o19) 2*p^2*cos(th)-2*p^2
(%i20) u_th:VP(p1-p4,p1-p4)
(%o20) -2*p^2*cos(th)-2*p^2
(%i21) t_th2:to_ao2(t_th,th)
(%o21) -4*p^2*sin(th/2)^2
(%i22) u_th2:to_ao2(u_th,th)
(%o22) 4*p^2*sin(th/2)^2-4*p^2
(%i23) t_thE:p_Em(t_th)
(%o23) 2*cos(th)*E^2-2*E^2-2*m^2*cos(th)+2*m^2
(%i24) u_thE:p_Em(u_th)
(%o24) -2*cos(th)*E^2-2*E^2+2*m^2*cos(th)+2*m^2
(%i25) load("MSQcomp.mac")
" MSQcomp() to compare matrix trace, symbolic trace\
  methods for square of polarized amplitudes\
  with square of spinor amplitudes. This is an\
  automated comparison, assuming A_spinor is\
  globally defined and also global sv1,sv2,sv3,sv4.\
  To see progress details, set details to true."
/* ===========================
  case RR --> LL
*/
(%i27) [sv1,sv2,sv3,sv4]:[1,1,-1,-1]$
(%i28) A_spinor : 2*m^2/p^2$
(%i29) MSQcomp();
" this comparison could take 3 plus minutes "

" agreement "
[sv1,sv2,sv3,sv4] = [1,1,-1,-1]
/*
================================
case RR --> RL
*/
(%i30) [sv1,sv2,sv3,sv4]:[1,1,1,-1]$
(%i31) A_spinor : m*sin(th/2)*E/(p^2*cos(th/2))-m*cos(th/2)*E/(p^2*sin(th/2))$
(%i32) MSQcomp();
" this comparison could take 3 plus minutes "

" agreement "
[sv1,sv2,sv3,sv4] = [1,1,1,-1]
(%i33) time(%);
(%o33) [199.12]
/*
=============================
case RR --> RR
*/
(%i34) [sv1,sv2,sv3,sv4]:[1,1,1,1]$
(%i35) A_spinor : m^2*sin(th/2)^2/(p^2*cos(th/2)^2)+2*sin(th/2)^2/cos(th/2)^2
                                +m^2*cos(th/2)^2/(p^2*sin(th/2)^2)
                                +2*cos(th/2)^2/sin(th/2)^2+4$
(%i36) MSQcomp();
" this comparison could take 3 plus minutes "

" agreement "
[sv1,sv2,sv3,sv4] = [1,1,1,1]
/*
```

```
==============================
case RL --> RL
*/
(%i37) [sv1,sv2,sv3,sv4]:[1,-1,1,-1]$
(%i38) A_spinor : m^2*cos(th/2)^2/(p^2*sin(th/2)^2)+
            2*cos(th/2)^2/sin(th/2)^2-m^2/p^2$
(%i39) MSQcomp();
" this comparison could take 3 plus minutes "

" agreement "
[sv1,sv2,sv3,sv4] = [1,-1,1,-1]
/*
===========================
case RL --> LR
*/
(%i40) [sv1,sv2,sv3,sv4]:[1,-1,-1,1]$
(%i41) A_spinor : m^2*sin(th/2)^2/(p^2*cos(th/2)^2)+
             2*sin(th/2)^2/cos(th/2)^2-m^2/p^2$
(%i42) MSQcomp();
" this comparison could take 3 plus minutes "

" agreement "
[sv1,sv2,sv3,sv4] = [1,-1,-1,1]
```

12.17 List of Dirac Package Files and Example Batch Files

The **Dirac package** consists of

- **dirac.mac**: This file loads in the other files and includes some utility functions and program startup settings.
- **dgexp.mac**: This file has code for symbolic expansions.
- **dgcon.mac**: This file has code for symbolic Lorentz index contraction.
- **dgtrace.mac**: This file has code for symbolic traces of products of gamma matrices.
- **dgeval.mac**: This file has code for frame dependent evaluation, such as **noncov**.
- **dgmatrix.mac**: This file has most of the code used for explicit Dirac spinor and matrix calculations.

The file **dgfunctions.txt** has an alphabetical listing of most of the Dirac package functions, together with a reference to the file in which the code can be found. There are twelve **example batch files** discussed in this chapter. In alphabetical order these are:

- **bhabha1.mac**: High energy electron-positron scattering.
- **bhabha2.mac**: Arbitrary energy electron-positron scattering.
- **compton0.mac**: Photon plus scalar charged particle scattering.
- **compton1.mac**: Photon plus Lepton scattering.
- **moller0.mac**: Scalar plus scalar charged particle scattering.
- **moller1.mac**: High energy limit of electron plus electron scattering.
- **moller2.mac**: Arbitrary energy electron plus electron scattering.
- **moller3.mac**: Use of explicit matrix trace methods and symbolic trace methods for the squared polarized amplitudes for arbitrary energy electron plus electron scattering.
- **pair1.mac**: Unpolarized two photon annihilation of an electron-positron pair.
- **pair2.mac**: Polarized two photon annihilation of an electron-positron pair.
- **photon1.mac**: Sum identities for photon polarization 3-vectors.
- **traceConEx.mac** demonstrates many of the most used package functions.

In addition to the above files, there is the file **MSQcomp.mac** which defines a function used by **moller3.mac**.