Computational Physics with Maxima or R: Ch. 1, Numerical Differentiation, Quadrature, and Roots [*]

Edwin (Ted) Woollett

August 31, 2015

Contents

1.1 Introduction . . . 3
1.2 Choice of R Interfaces and Elements of the R Language . . . 7
1.2.1 The R Function curve for a function or expression plot . . . 14
1.3 Elements of the Maxima Language . . . 18
1.3.1 The Maxima Function plot2d . . . 22
1.4 Numerical Derivatives . . . 22
1.4.1 Numerical Derivative Functions in R . . . 23
1.4.2 Testing Simple Numerical Derivative Methods in R . . . 27
1.4.3 Testing Simple Numerical Derivative Methods in Maxima . . . 29
1.5 Numerical Quadrature . . . 33
1.5.1 R function integrate for one dimensional integrals . . . 33
1.5.2 R function elliptic::myintegrate for integration of a complex function . . . 34
1.5.3 R function cubature::adaptIntegrate for multi-dimensional quadrature . . . 34
1.5.4 Maxima quadrature functions quad_qags and quad_qagi . . . 37
1.5.5 Trapezoidal Rule for a Uniform Grid in R . . . 38
1.5.6 Trapezoidal Rule for a Uniform Grid in Maxima . . . 40
1.5.7 Trapezoidal Rule for a Non-Uniform Grid in R . . . 42
1.5.8 Trapezoidal Rule for a Non-Uniform Grid in Maxima . . . 43
1.5.9 Simpson's 1/3 Rule in R . . . 44
1.5.10 Simpson's 1/3 Rule in Maxima . . . 46
1.5.11 Checking Integrals with the Wolfram Alpha Webpage . . . 47
1.5.12 Dealing with an Integral with Infinite or Very Large Limits . . . 48
1.5.13 Handling Integrable Singularities at Endpoints . . . 51
1.6 Finding Roots . . . 53
1.6.1 R Function uniroot . . . 53
1.6.2 R function polyroot . . . 54
1.6.3 R function rootSolve::uniroot.all . . . 54
1.6.4 R Newton-Raphson: newton . . . 55
1.6.5 R function secant . . . 56
1.6.6 Maxima functions find_root and bf_find_root . . . 57
1.6.7 Maxima Newton-Raphson: newton . . . 59
1.6.8 Maxima Bigfloat Newton-Raphson: bfnewton . . . 60
1.6.9 Maxima function secant . . . 61
1.6.10 Divide and Conquer Root Search . . . 62
1.7 mtext and Plot Margins in R . . . 63
1.8 Drawing Circles with R . . . 67
1.8.1 Homemade Circles in R . . . 67
1.8.2 Circles Using the R Package shape . . . 68
1.8.3 Circles Using the R Function symbols . . . 70

[*]The code examples use **R ver. 3.0.1** and **Maxima ver. 5.28** using **Windows XP**. This is a live document which will be updated when needed. Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions for improvements to `woollett@charter.net`

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

1.1 Introduction

These notes focus on the areas of computational physics found in the textbook **Computational Physics** by Steven E. Koonin (see below). Many of the exercises, examples, and projects which appear in this text are discussed in some detail. In these notes, small program codes will be included in the text. The much larger example and project codes will be found as separate text files with names such as exam1.R, exam1.mac, proj1.R, proj1.mac etc., on the author's webpage (see footnote on the table of contents page).

Examples of simple numerical methods are shown using both **R** and **Maxima**, in parallel sections. It is often useful to have simple homemade code for an initial exploration since it is easy to include debug printouts (when useful) to isolate the gory details of what is at issue.

Understanding how to code simple numerical methods is also an excellent way to learn the basic syntax of the **R** and **Maxima** languages. Each of these software programs have their own unique and special attributes and advantages, and one can make faster progress by having simultaneously a Maxima window open and a **R** window. In developing substantial programs for the major examples and projects, use is made of the built-in core functions and package functions which have the highest accuracy and efficiency to the extent such are available.

We use the powerful (and free) **R** language software (**http://www.r-project.org/**) which comes with the default graphics interface **RGui**, and also use the (also free) integrated development environment (IDE)
RStudio (**http://www.rstudio.com/**).

The **R** packages **deSolve**, **rootSolve**, **bvpSolve**, and **ReacTran** are the primary (open source) R packages used and their existence is the primary reason for choosing to devote equal space to the use of what has (in the past) been primarily a framework for complicated statistical calculations and models.

We also use, in parallel treatments of each topic, the free and open source Maxima computer algebra system (CAS) software **http://maxima.sourceforge.net/**. In particular, Maxima is used whenever symbolic integrals, derivatives, limits, symbolic solutions of sets of algebraic or differential equations, or symbolic simplifications of expressions are needed in setting up the code of an example or project. Maxima has its own suite of numerical abilities and is especially useful when one needs arbitrary precison calculations.

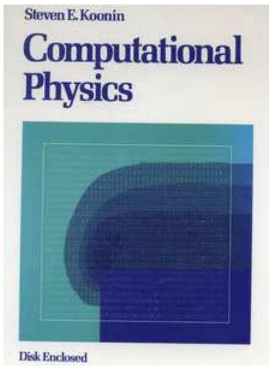

Figure 1: Computational Physics, 1986, Basic Language

The landmark text **Computational Physics**, by Steven E. Koonin, was published in 1986 by The Benjamin/Cummings Publishing Company, Inc, and used the Microsoft version of the BASIC language designed to run on IBM (and clone) personal computers.

A version with Fortran code (with the coauthor Dawn C. Meredith) was published in 1990 by Perseus Books. The following description of the Fortran edition appears on the Amazon website:

> Computational Physics is designed to provide direct experience in the computer modeling of physical systems. Its scope includes the essential numerical techniques needed to "do physics" on a computer. Each of these is developed heuristically in the text, with the aid of simple mathematical illustrations. However, the real value of the book is in the eight Examples and Projects, where the reader is guided in applying these techniques to substantial problems in classical, quantum, or statistical mechanics. These problems have been chosen to enrich the standard physics curriculum at the advanced undergraduate or beginning graduate level. The book will also be useful to physicists, engineers, and chemists interested in computer modeling and numerical techniques. Although the user-friendly and fully documented programs are written in FORTRAN, a casual familiarity with any other high-level language, such as BASIC, PASCAL, or C, is sufficient. The codes in BASIC and FORTRAN are available on the web at http://www.computationalphysics.info. They are available in zip format, which can be expanded on UNIX, Window, and Mac systems with the proper software. The codes are suitable for use (with minor changes) on any machine with a FORTRAN-77 compatible compiler or BASIC compiler. The FORTRAN graphics codes are available as well. However, as they were originally written to run on the VAX, major modifications must be made to make them run on other machines.

Although both of these versions are now out of print, used copies are available via the web (for example at Amazon.com) for as little as ten dollars for the Fortran version and six dollars for the Basic version. The reader of these notes will gain more understanding of the examples worked out here (using Maxima or the R language) by also reading either of the Koonin texts. For the "Examples" and "Projects", (and for the average reader) the Basic code will probably be easier to follow than the Fortran code.

Westview Press, part of the Perseus Books Group, at the address:
`http://www.westviewpress.com/book.php?isbn=9780201386233`
offers the following prices (available in lightening print mode):

```
Computational Physics Fortran Version, by Steven E. Koonin,
August 1998 Trade Paperback, 656 Pages, $77.00 U.S.,
  $88.99 CAN, 51.99 U.K., 54.99 E.U., ISBN 9780201386233
```

When you click the buy icon on the Westview Press webpaqe, you are offered a list of online book sellers, such as Amazon.com, and a list of local booksellers, such as Barnes and Noble.

The booksamillion.com website at
`http://www.booksamillion.com/product/9780201386233#overview`
offers the new paperback for **`$77`**:

```
ISBN-13: 9780201386233,  ISBN-10: 0201386232, Westview Press
July 1998, 656 pages
```

The fortran codes and the Basic codes are available as separate zip files at
`http://www.computationalphysics.info/`.

From the section "How to use this book" in the Basic version:

> This book is organized into chapters, each containing a text section, an example, and a project. Each text section is a brief discussion of one or several related numerical techniques, often illustrated with simple mathematical examples ...
>
> The example and project in each chapter are applications of the numerical techniques to particular physical problems. Each includes a brief exposition of the physics, followed by a discussion of how the numerical techniques are to be applied.
>
> The examples and projects differ only in that the student is expected to use (and perhaps modify) the program which is given for the former in Appendix B while the book provides guidance in writing programs to treat the latter through a series of steps...
>
> However, programs for the projects have been included in Appendix C; these can serve as models for the student's own program or as a means of investigating the physics without having to write a major program "from scratch". A number of suggested studies accompany each example and project; these guide the student in exploiting the programs and understanding the physical principles and numerical techniques involved.

The table of contents of Koonin's **Computational Physics** follows:

```
Chapter 1: Basic Mathematical Operations
   1.1 Numerical Differentiation
   1.2 Numerical Quadrature
   1.3 Finding Roots
   1.4 Semiclassical quantization of molecular vibrations
   Project I: Scattering by a central potential

Chapter 2: Ordinary Differential Equations
   2.1 Simple Methods
   2.2 Multistep and implicit methods
   2.3 Runge-Kutta methods
   2.4 Stability
   2.5 Order and chaos in two-dimensional motion
   Project II: The structure of white dwarf stars
      II.1 The equations of equilibrium
      II.2 The equation of state
      II.3 Scaling the equations
      II.4 Solving the equations

Chapter 3: Boundary Value and Eigenvalue Problems
   3.1 The Numerov algorithm
   3.2 Direct integration of boundary value problems
   3.3 Green's function solution of boundary value problems
   3.4 Eigenvalues of the wave equation
   3.5 Stationary solutions of the one-dimensional Schroedinger equation
   Project III: Atomic Structure in the Hartree-Fock approximation
      III.1 Basis of the Hartree-Fock approximation
      III.2 The two-electron problem
      III.3 Many-electron systems
      III.4 Solving the equations

Chapter 4: Special Functions and Gaussian Quadrature
   4.1 Special functions
   4.2 Gaussian quadrature
   4.3 Born and eikonal approximations to quantum scattering
   Project IV: Partial wave solution of quantum scattering
      IV.1 Partial wave decomposition of the wavefunction
      IV.2 Finding the phase shifts
      IV.3 Solving the equations

Chapter 5: Matrix Operations
   5.1 Matrix inversion
   5.2 Eigenvalues of a tri-diagonal matrix
   5.3 Reduction to tri-diagonal form
   5.4 Determining nuclear charge densities
   Project V: A schematic shell model
      V.1 Definition of the model
      V.2 The exact eigenstates
      V.3 Approximate eigenstates
      V.4 Solving the model

Chapter 6: Elliptic Partial Differential Equations
   6.1 Discretization and the variational principle
   6.2 An iterative method for boundary-value problems
   6.3 More on discretization
   6.4 Elliptic equations in two dimensions
```

Project VI: Steady-state hydrodynamics in two dimensions
VI.1 The equations and their discretization
VI.2 Boundary Conditions
VI.3 Solving the equations

Chapter 7: Parabolic Partial Differential Equations
7.1 Naive discretization and instabilities
7.2 Implicit schemes and the inversion of tri-diagonal matrices
7.3 Diffusion and boundary value problems in two dimensions
7.4 Iterative methods for eigenvalue problems
7.5 The time-dependent Schroedinger equation
Project VII: Self-organization in chemical reactions
VII.1 Description of the model
VII.2 Linear stability analysis
VII.3 Numerical solution of the model

Chapter 8: Monte Carlo Methods
8.1 The basic Monte Carlo strategy
8.2 Generating random variables with a specified distribution
8.3 The algorithm of Metropolis et al.
8.4 The Ising model in two dimensions
Project VIII: Quantum Monte Carlo for the H2 molecule
VIII.1 Statement of the problem
VIII.2 Variational Monte Carlo and the trial wavefunction
VIII.3 Monte Carlo evaluation of the exact energy
VIII.4 Solving the problem

Figure 2: Steven E. Koonin

Steven Koonin has held positions in both business and government since leaving Cal Tech, and has recently returned to academia. A New York University press release recently announced the appointment of Koonin as the Director of the new

Center for Urban Science and Progress (CUSP). Here is a partial quote from that announcement:

> Dr. Koonin was confirmed by the Senate in May, 2009 as Undersecretary for Science at the U.S. Department of Energy, serving in that position until November, 2011. Prior to joining the Obama Administration, he was BPs Chief Scientist, where he was a strong advocate for research into renewable energies and alternate fuel sources. He came to BP in 2004 following almost 3 decades as Professor of Theoretical Physics at the California Institute of Technology, serving as the Institute's Vice President and Provost for the last nine years. Koonin comes to CUSP most immediately from a position at the Science and Technology Policy Institute of the Institute for Defense Analyses in Washington, DC.
>
> Koonin's research interests have included nuclear astrophysics; theoretical nuclear, computational, and many-body physics; and global environmental science. He has been involved in scientific computing throughout his career. He has supervised more than 30 PhD students, produced more than 200 peer-reviewed research publications, and authored or edited 3 books, including a pioneering textbook on Computational Physics in 1985. As Caltech's Provost, Koonin oversaw its research and educational programs, including the hiring of one-third of the Institute's faculty. At BP, he conceived and established the Energy Bioscience Institute at UC Berkeley and the University of Illinois, while at DOE, he led the preparation of both its most recent Strategic Plan and its first Quadrennial Technology Review for energy. Koonin has served as an advisor for numerous academic, government, and for-profit organizations.
>
> Dr. Koonin is the recipient of numerous awards and honors, including the George Green Prize for Creative Scholarship at Caltech, a National Science Foundation Graduate Fellowship, an Alfred P. Sloan Foundation Fellowship, and a Senior U.S. Scientist Award (Humboldt Prize), and the Department of Energy's Ernest Orlando Lawrence Award. He is a Fellow of several professional societies, including the American Physical Society, the American Association of the Advancement of Sciences, and the American Academy of Arts and Sciences, and a member of the Council on Foreign Relations and the U.S. National Academy of Sciences.

A good 3 minute YouTube video of Steven Koonin summarizing his critique of cold fusion at a science conference in Baltimore in 1989 is available at **`http://www.youtube.com/watch?v=wR-AohRWbBo`**. The author enjoyed a similar presentation by Koonin at the weekly Physics Department Colloquium at California State University at Long Beach in that same year.

1.2 Choice of R Interfaces and Elements of the R Language

If you are new to the **`R`** language, you should first download and install the latest (free and open source) Windows version of **`R`** from **`http://www.r-project.org/`**, choosing the default answers for all questions asked during the install process. You should see the **`R`** icon on your desktop. Clicking that icon will start up the default user interface, a Windows IDE called **`RGui`**, with a blinking cursor in the **`R Console`** window. This interface will allow you to start getting familiar with the **`R`** language.

You can customize the look of the Console window using the menu bar **`Edit, GUIpreferences...`**,

The author's settings are 1. checked boxes: MDI, MDI Toolbar, multiple windows, and True Type only, 2. Font set to Courier New, Font size set to 12, Font style set to bold, 3. Console rows = 25, Console columns = 80, 4. background color = wheat1, normal text color = black, user text color = blue, pagerbg color = white.

Keyboard Shortcuts

Select **`Help, Console`** to get a window of keyboard shortcuts. One of the most useful is the two key command **`Ctrl+Tab`** to toggle between the Console window and the graphics window ("device"). Another keyboard shortcut to get back to the Console window (from a graphics window, for example,) is **`(Alt+w, 1)`**, which makes use of the **`Windows`** submenu.

To clear the screen and move the cursor to the top of the Console window, use the keyboard shortcut **`Ctrl+l`** (lowercase of letter L), or else use **`(Edit, Clear console)`**.
The up-arrow and down-arrow keys cycle through the "command history". The up-arrow key of the keyboard only brings in one line at a time from the previous inputs, which is not convenient to bring into action a multiline input. It is also very difficult to edit a multiline command inside the **`RGui`** Console window. Hence the author recommends that multiline

input commands be set up in a text editor (containing a daily "work diary", for example) such as Notepad2 or Notepad++ which have strong parenthesis and bracket matching features, and be pasted into the Console (using **`Ctrl+v`**) to try out the current code. You can set up multiple function definitions and parameter assignments, copy the whole group to the clipboard (using **`Ctrl+c`**), and then paste the whole group into the console at once using **`Ctrl+v`**.

Getting work done in the Console window is a little primitive. The **`INS`** key toggles the overwrite mode on and off (initially off). When editing a line in the Console, you can use the **`Home`**, **`End`**, **`left-arrow`**, and **`right-arrow`** keys to move left and right. You can use either the **`backspace`** and/or the **`delete`** keys to delete characters. You *cannot* use **`Shift+right-arrow`** to highlight a set of contiguous characters. You *cannot* use **`Shift+End`** to highlight part or all of a line. However, **`Ctrl+Del`** will delete from the current character to the end of the current line, and **`Ctrl+U`** will delete all text from the current line.

Clicking on the Console window does not move the cursor. **`Ctrl+Home`** and **`Ctrl+End`** do nothing. **`Page-Up`** and **`Page-Down`** do nothing. To copy only some lines of input and output for transfer back to your text based work file or to a LaTeX document, you must drag the cursor over the lines desired to highlight, and then use **`Ctrl+c`** to copy to the clipboard.

You can copy *all* the current Console windows contents (i.e., since the last use of **`Ctrl+l`**) using **`(Edit, Select All)`**, and then either **`(Edit, Copy)`** or **`Ctrl+C`**. To turn off the selection highlight, you can either press the **`Backspace`** key or use the mouse to click in the Console window.

Use repeated down-arrow's to restore the cursor to the interpreter prompt. Pressing the **`Esc`** key will also restore the interpreter prompt (and will also stop the interpreters current action).

Entering, for example, **`?curve`** will launch a full page browser view of documentation for the **`R`** function **`curve`**, a view mode which is easier to read and copy from compared with the rather constricted help panel in the RStudio environment (see below). Entering just **`curve`** without the leading question mark, and without any **`(...)`** will display the **`R`** code which defines the function.

To quit **`R`**, use **`q()`**, (followed as usual by **`Enter`**). You will be asked if you want to save the current state of the work and settings, which requires pressing either the letter **`n`** or **`y`**.

Your current working directory is given by **`getwd()`**. Change your current working directory with **`setwd("c:/k1")`**, for example, or else use **`(File, Change dir...)`**. The author sets his chosen working folder (directory) via his startup file. The most convenient place to put code commands you want to always be loaded upon startup of **`R`** is in the text file **`Rprofile.site`** which has the path (on the author's computer)
`C:\Program Files\R\R-3.0.1\etc\Rprofile.site`, (more about this file later).

You can see what files are in your current working directory with **`list.files`**, get file information with **`file.info`**, and view file contents with **`file.show`**.

```
> getwd()
[1] "c:/k1"
> list.files()
 [1] "cp1.aux"          "cp1.dvi"          "cp1.log"
 [4] "cp1.tex"          "cp1.toc"          "cpnewton.mac"
 [7] "mycurve.eps"      "mycurve.jpg"      "mycurve.png"
[10] "mycurve1.eps"     "mycurve2.eps"     "new-timedate.lisp"
[13] "test1.eps"        "test2.eps"        "work2013-10-09.txt"
[16] "XMaxima-5-28.lnk"
> file.info("cpnewton.mac")
             size isdir mode               mtime               ctime
cpnewton.mac 2082 FALSE  666 2013-10-07 11:25:59 2013-10-09 12:48:07
                            atime exe
cpnewton.mac 2013-10-10 12:03:59  no
```

```
> file.show("cpnewton.mac")
```

The last command (**file.show**) opened up a separate window called the R Information window, where the contents of the file were displayed.

The **cat** function in the **RGui** Console does not automatically add a "line advance" in interactive use (as does **RStudio**). Instead you need to add a string **"\n"** as the last element:

```
> a=2; b = 4; c = 6
> cat(a,b,c)
2 4 6> cat(a,b,c,"\n")
2 4 6
```

Depending on your aptitude for dealing with a busy environment, you might wish to download (from **http://www.rstudio.org/** the (free and open source) **RStudio** integrated development environment (IDE), which makes it easier to keep track of what objects are currently known in the workspace, which packages are currently loaded, and has easily seen windows for help documentation requested, plots made, a window of a history of commands used in interactive use, and a window displaying the files in the current folder. And the submenu brought up via the Session menu choice makes it easy to set the "working directory" (working folder). There is also a text window (called Source) for writing and editing code which has an easy method for pasting into the Console window.

The Console window of RStudio is the place to either type in or paste in (from either the Source window or from a text editor such as the free and very useful **Notepad2** (**http://sourceforge.net/projects/notepad2/**). Pressing **Enter** at the end of such Console input will cause **R** to interpret and "run" the command(s).

You can enlarge the width of the Console window by dragging on the right hand border of the Console window. However, if you try to expand the width of the Console window too far, the present behavior suddenly pushes the normal right window panes out of sight, and you must do some serious dragging from the right edge to recover the normal multiple pane view; there is apparently no keyboard command which will restore the default view and size of the **RStudio** windows.

To return the cursor to the Console window, when in any other RStudio window, use the two-key command **Ctrl+2**. There are fast keyboard shortcuts for most things. To clear the Console screen and place the cursor at the top of the Console window, use **Ctrl+l** (that is lower case letter **l**, not number **1**). To quit **R**, use **q()**, (followed as usual by **Enter**).

If you happen to choose (from the munu) **(File, Close All)**, the Source window will vanish. You can restore an altered version using either: **(File, New, Rscript)**, or **File, New, Text File**, or **File, Open ....** Then you can return the cursor to the Source window using **Ctrl+1** (number **1**).

You should then go to the **RStudio** documentation page **http://www.rstudio.com/ide/docs/** and work through the nine sections of tutorial material under the heading **Using RStudio**, with the following sections: Working in the Console, Editing and Executing Code, Code Folding and Sections, Navigating Code, Using Projects, Command History, Working Directories and Workspaces, Customizing RStudio, Keyboard Shortcuts. The sections on Code Folding, Projects and Customizing are of very secondary interest to a new user.

A somewhat controversial subject is whether one should restrict oneself to the notation **<-** for "assignment" in **R**, or whether it is kosher to use instead the equal sign **=** in place of **<-**. Thus **a <- 3.2** or **myf <- function(x,y) {x*sin(y)}** are respectively an assignment of the floating point number **3.2** to the symbol **a**, and the assignment of a function definition to the symbol **myf** , the latter used with the syntax **b <- myf(2,1.2)**, for example.

Either **<-** or **=** will result in correctly behaving code in practice. Purists in **R** denigrate the use of the equal sign **=** for the assignment operation. Physicists are used to choosing the easiest method of getting the job done. The author prefers the equal sign for assignment because it requires one keyboard operation, rather than two, and also because the resulting code can be more speedily visually scanned. The **<-** notation slows the visual scan speed, because one needs to be sure a minus sign was not inadvertently inserted at a position not intended. Thus the author uses **a = 3.2** and **myf = function(x,y) {x*sin(y)}**.

Miscellaneous Tips on R

R treats the symbols **n** and **N** as distinct. (Case matters.)

The author prefers to expand the **Console** window (of **RStudio**) both vertically and horizontally (the latter by dragging the window edge, but not too large!). When you then make a plot, the plot window will become automatically visible, but will be small. You can either drag the edge of the plot window to the left, to see more clearly, and/or you can click on the zoom icon near the top of the plot window, thus creating a much larger and dedicated window for the plot.

The author prefers to design code in a text editor such as **Notepad2** (see above mention) partly because the editor uses strong color contrast for matching parentheses and brackets, and it is easy to see if you have missed some closing parentheses or brackets. You can select one or more **R** statements and paste them into the **Console** as long as the end of each line of code in clearly incomplete or clearly complete. If a line of code in the text editor needs completion on the next line, split the line after a comma, for example so that the **R** interpreter knows more must be coming.

The most convenient place to put code commands you want to always be loaded upon startup of **R** is in the text file **Rprofile.site** which has the path **C:\Program Files\R\R-3.0.1\etc\Rprofile.site**.

At the moment, the author's **Rprofile.site** file contains

```
## %% is the mod function
is.even = function(x) x %% 2 == 0
is.odd = function(x) x %% 2 != 0
##  symbolic derivatives, default is first derivative
##  uses built-in D function
DD = function(expr, name, order = 1) {
  if(order < 1) stop("'order' must be >= 1")
   if(order == 1) D(expr, name)
   else DD(D(expr, name), name, order - 1)}
# example:
#  > DD(expression(sin(x)^2),"x")
#    2 * (cos(x) * sin(x))
ps = function(filename) {
         postscript(file=filename ,paper="special",
         width=10,height=10,onefile=F,horizontal=F)}
setwd("c:/k1")
library(deSolve)
library(rootSolve)
```

The last two commands load the two packages **rootSolve** and **deSolve**. The symbol **#** is the comment symbol in **R**; **R** ignores the remainder of that line.

You can find what **R** considers your "home" folder by interactively using two commands

```
> setwd("~/")

> getwd()
[1] "C:/Documents and Settings/Edwin Woollett/My Documents"
```

and that is where **R** looks for a possible second startup text file called **.Rprofile**, which you can use in place of (or together with) the file **Rprofile.site** mentioned above.

You can get help in the **RStudio** help panel on any known **R** object using, for example

```
> ?getwd
```

and the **RStudio** help panel will display the documentation.

Numbers in **R** are normally considered floating point numbers. If you want to define a true integer, use the letter **L** after the integer, as we do here:

```
> a = 2
> is.integer(a)
[1] FALSE
> b = 2L
> is.integer(b)
[1] TRUE
```

You can also generate true integers using the **from:to** notation or the **seq(from, to, by)** notation.

```
> xv = 1:3
> is.integer(xv)
[1] TRUE
> yv = seq(1, 3)
> is.integer(yv)
[1] TRUE
```

R does not automatically show you what you have produced with an assignment; this is often an advantage, but also often you want to double-check your assignment operation by immediately looking at it. There are two ways to do this on one line , first by surrounding the assignment statement with opening and closing parentheses, second by ending the assignment with a semicolon **;** and typing the name of the object:

```
> (yv = sin(xv) )
[1] 0.8414710 0.9092974 0.1411200
> yv = cos(xv); yv
[1]  0.5403023 -0.4161468 -0.9899925
```

The author prefers the semicolon method and doesn't like to stare at the extra parentheses in the first method. Note, also, that it is ok to place several assignment statements on the same line, and then displayed using **cat** (done in **RStudio**, so the extra line advance string **"\n"** does not have to be included as the last element of **cat**):

```
> a = 2.2; b = 3.4; c = 4.5
> cat(a,b,c)
2.2 3.4 4.5
```

The function **cat()** is especially useful for inserting debug printouts in your code. When used interactively, as in:

```
> cat("a = ",a," b = ",b," c = ",c)
a =  2.2  b =  3.4  c =  4.5
```

within the **RStudio** Console window, an automatic line advance is issued when a **cat** statement occurs. For pretty display of results **inside** a function definition, you would include at the end of each **cat** a line advance string as in **"\n"** (read this as "escape, n") or **"\n\n"**, the latter resulting in a blank line in addition to the line advance.

However, in setting up **debug printouts**, it pays to compress the output into a smaller space on the console, so the author deliberately leaves out the line advance string. Here is an example. In **Notepad2** the following meaningless function is defined

```
mytest = function() {
     xv = seq(0,1,by = 0.25)
     yv = sin(xv)
     cat(" xv = ",xv," yv = ",yv)
     zv = exp(yv)
     cat(" zv = ",zv)
     xv = (1:4)/4
     cat(" xv = ", xv)
     yv = sin(xv)
     cat( " yv = ", yv ," \n\n")}
```

as a demonstration of what happens when you leave out the line advance string at the end of all but the last **cat**. The above code was selected and copied to the Windows Clipboard, and then pasted into the **RStudio** console, and then the function was tried out. **R** adds a **+** sign to the beginning of each successive line, indicating it knows the code fragment is not yet complete (since the final closing bracket has not yet been found).

```
> mytest = function() {
+     xv = seq(0,1,by = 0.25)
+     yv = sin(xv)
+     cat(" xv = ",xv," yv = ",yv)
+     zv = exp(yv)
+     cat(" zv = ",zv)
+     xv = (1:4)/4
+     cat(" xv = ", xv)
+     yv = sin(xv)
+     cat( " yv = ", yv ," \n\n")}
> mytest()
 xv =  0 0.25 0.5 0.75 1  yv =  0 0.247404 0.4794255 0.6816388 0.841471 zv =  1 1.280696
1.615146 1.977115 2.319777 xv =  0.25 0.5 0.75 1 yv =  0.247404 0.4794255 0.6816388 0.841471
```

(If you try this in the default **RGui** interface, the output is on one very long line, so you must scroll the screen rightward to see everything, and the visible portion ends with a **$** to warn you there is more. But if you drag and copy the screen, you get the whole output.)

With both interfaces, if you copy the screen and paste into a **verbatim** section of a LaTeX file, after running LaTeX, the **dvi** file will show only part of the long line; you must line break manually in the LaTeX file.

R, by default, displays 7 digits for floating point numbers (while internally using 16 digit arithmetic). To make the compressed debug output even more convenient, we can use the **options(digits = m)** command, and re-run the function:

```
> options(digits = 3)
> mytest()
 xv =  0 0.25 0.5 0.75 1  yv =  0 0.247 0.479 0.682 0.841 zv =  1 1.28 1.62 1.98 2.32 xv =  0.25
 0.5 0.75 1 yv =  0.247 0.479 0.682 0.841
```

The function **ls()** displays the names of the named objects in your current workspace.

```
> ls()
[1] "a"      "b"      "c"      "mytest" "xv"
[6] "yv"
```

Notice that **zv** is not known; variables assigned values inside a function are local to that function and are not known in the global environment.

You can remove all currently known named objects from memory using

```
> rm(list = ls())
> ls()
character(0)
```

We can use **data.frame** to make a table with headings.

```
> options(digits = 5)
> xv = (1:10)/10; yv = sin(xv); zv = cos(xv)
> xyz = data.frame(x = xv, sin = yv, cos = zv)
> xyz
     x      sin     cos
1  0.1 0.099833 0.99500
2  0.2 0.198669 0.98007
3  0.3 0.295520 0.95534
4  0.4 0.389418 0.92106
5  0.5 0.479426 0.87758
6  0.6 0.564642 0.82534
7  0.7 0.644218 0.76484
8  0.8 0.717356 0.69671
9  0.9 0.783327 0.62161
10 1.0 0.841471 0.54030
```

The dollar sign **$** suffix is used to pick out parts of a data frame for later use, using **object-name$part-name**:

```
> xyz$x
 [1] 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
> xyz$sin
 [1] 0.099833 0.198669 0.295520 0.389418 0.479426 0.564642 0.644218 0.717356
 [9] 0.783327 0.841471
```

R has no single precision data type. All real floating point numbers are stored in double precision format. Hence the coercive functions **as.numeric()**, **as.double()**, and **as.single()** all do the same job.

A vector in **R** is not necessarily a physical vector, but is an ordered collection of objects of the same type. **xv = 1:10** defines **xv** as a vector, and so does **xv = seq(1,10)**, and so does **xv = c(1,2,3,4,5,6,7,8,9,10)**. The **c()** function is one of the most used **R** functions, and the ‘c’ stands for ‘concatenate’. Individual elements of a vector **xv** are accessed using a single bracket, as in **xv[3]**.

```
> xv = 1:3; yv = seq(1,3,by = 0.5); zv = c(1.2, 2.3, 4.5)
> cat(xv[2], yv[4], zv[1])
2 2.5 1.2
```

Note that the **R** function **sin** turns vectors into vectors and single numbers into single numbers, and so does **xˆ2**, etc.

```
> yv = seq(0,1, by = 0.25)
> yv
[1] 0.00 0.25 0.50 0.75 1.00
> f1 = function(xv) xvˆ2
> f1(yv)
[1] 0.0000 0.0625 0.2500 0.5625 1.0000
> sin(yv)
[1] 0.0000000 0.2474040 0.4794255 0.6816388
[5] 0.8414710
> f1(3)
[1] 9
> sin(3)
[1] 0.14112
```

Some vector arithmetic examples (note you can divide by a vector!):

```
> x = c(2, 4, 6); y = c(2, 2, 2)
> x*y
[1]  4  8 12
```

```
> x/y
[1] 1 2 3
> x^2/y
[1]  2  8 18
> sin(x)/y
[1]  0.4546487 -0.3784012 -0.1397077
> 1/x
[1] 0.5000000 0.2500000 0.1666667
```

Use **head** and **tail** to look at the first few and last few elements of a long vector or data frame. In this example we use **rnorm(num, mean, stand-dev)**. If you use the entry **?rnorm**, a help panel explains that this function generates random numbers from a "normal" (or "gaussian") distribution, and that the names and default values for the args are: **rnorm(n, mean = 0, sd = 1)**. See **http://en.wikipedia.org/wiki/Normal_distribution** for detailed information.

```
> x = rnorm(50, 0, 1)
> head(x)
[1] -1.5915329  0.5068380  0.6856412 -0.7527810 -0.4271025  0.9496289
> tail(x)
[1]  0.53163641  0.01183301 -0.65729463  0.55093746 -1.03067796 -0.52047072
```

1.2.1 The R Function curve for a function or expression plot

The syntax for **curve** is

```
curve(expr, from = NULL, to = NULL, n = 101, add = FALSE,
      type = "l", xname = "x", xlab = xname, ylab = NULL,
      log = NULL, xlim = NULL, ...)
```

In this definition, **expr** is the name of a function, or a call or an expression written as a function of x which will evaluate to an object of the same length as x.

The command **curve(x^2,-3,3)** produces an expected plot, but not:

```
> curve(x,-3,3)
Error in eval(expr, envir, enclos) : could not find function "x"
```

Use of the "expression" **x** alone in the first slot causes **R** to look for a named function in the environment with the name **x**, and not even

```
> curve(z,-3,3,xname = "z")
Error in eval(expr, envir, enclos) : could not find function "z"
```

overcomes this isolated problem with **curve**.

One can use **abline(0,1,...)** to add a straight line with y-intercept **0** and slope **1** to deal with this case, as in (the optional arg **type = "n"** to **plot** creates a blank canvas with the specified horizontal and vertical ranges):

```
> plot(-3:3, -3:3, type = "n",xlab = "", ylab = "")
> abline(h=0,lwd=2)
> abline(v=0,lwd=2)
> abline(0, 1, lwd = 4, col = "blue")
```

but beware: in **`plot(x,y,...)`** the vectors **`x`** and **`y`** must be the *same* length.

Use of either **`curve(x^2 - 5, -3, 3)`** or **`curve(y^2 - 5,-3,3,xname = "y")`** will draw the same figure. The value of **`xname`** defaults to **`"x"`**.

You can increase the number of function samples taken for the plot by overriding the default value of **`n`**, for example with **`curve(x^2-5,-3,3,200)`**, or by using named arguments, avoiding respecting the order of the arguments: **`curve(x^2-5,-3,3,lwd=3,col = "blue",n = 200)`**.

The **`lwd`** parameter defaults to **`lwd = 1`**, so using **`lwd=3`** makes the line three times as thick. The default line type for **`curve`** is a solid line (**`lty = 1`**), and the default color is **`col = "black"`**.

The **`R`** functions **`plot`**, **`matplot`**, **`points`**, **`abline`**, and **`lines`** can each deal with additional args involving **`lty`**, **`lwd`**, **`col`**, and **`type`**.

To avoid the default labels on the horizontal ("x") axis and the vertical ("y") axis, you can include **`xlab = ""`** and **`ylab = ""`**, or else furnish your own string, such as **`xlab = "time (sec)"`**.

The default behavior of **`curve`** (add = FALSE) starts a new figure, ignoring any past graphics commands. If you want to add additional elements to your initial plot, the "high-level" function (such as **`curve`** or **`plot`**) needs to be invoked with the extra arg **`add = TRUE`**. In contrast, using any of the "low-level" functions **`points`**, or **`abline`** or **`lines`** simply adds to your initial plot.

Here is an example of building up a figure with a grid, x and y axes, and three curves, including a straight line:

```
> curve(x^3,-3,3,lwd=3,col="green",xlab="",ylab="")
> grid(lty="dashed",col="black")
> abline(h=0,lwd=2)
> abline(v=0,lwd=2)
> curve(x^2,-3,3,add=TRUE,lwd=3,col="blue")
> abline(0,1,lwd = 3, col = "red")
```

which produces:

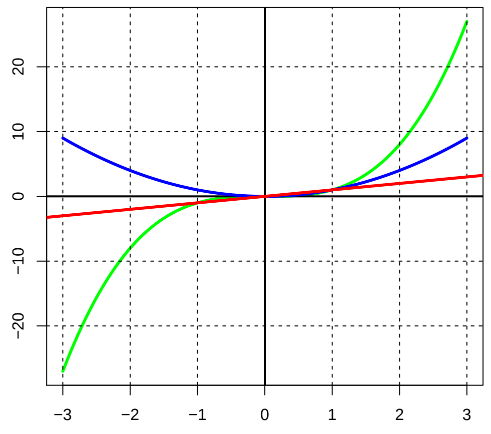

Figure 3: first curve example

Here is a second example of building up a figure with three uses of **`curve`**:

```
> curve(x^2-5,-5,5,col="blue",lwd=3,xlab="",ylab="",ylim=c(-10,25))
> grid(lty="dashed",col="black")
> abline(v=0,lwd=2)
> abline(h=0,lwd=2)
> curve(y^3, -3, 3, add = TRUE, lwd=3, xname = "y")
> curve(y^5, -3, 3, add = TRUE, lwd=3, xname = "y",col="red")>
```

which produces

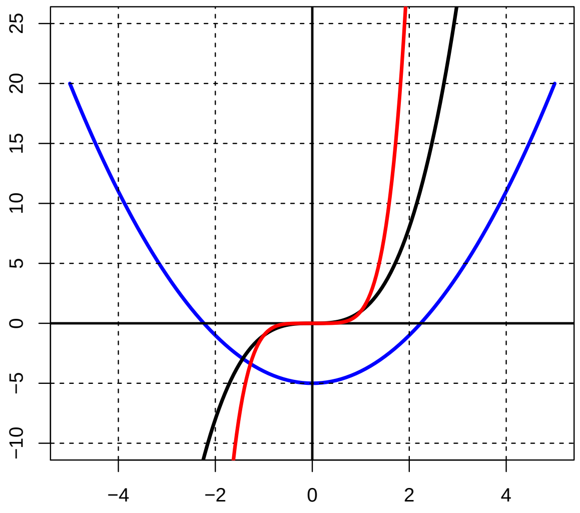

Figure 4: second curve example

Using curve with Piecewise Defined Functions

Here are two examples of using **`curve`** with piecewise defined functions.

```
> f=function(x) ifelse(x<=1,1+1/x,1/x)
> curve(f,0.5,3,lwd=3,col="blue",n=301)
```

which produces the plot:

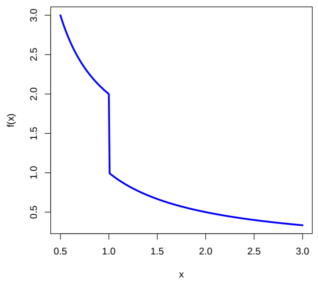

Figure 5: Two Section Plot

and

```
> f=function(x){
+   ifelse(x<=1,2+1/x,
+     ifelse(x<=2,1+1/x,1/x))}
> curve(f,0.5,3,lwd=3,col="blue",n=301)
```

which produces

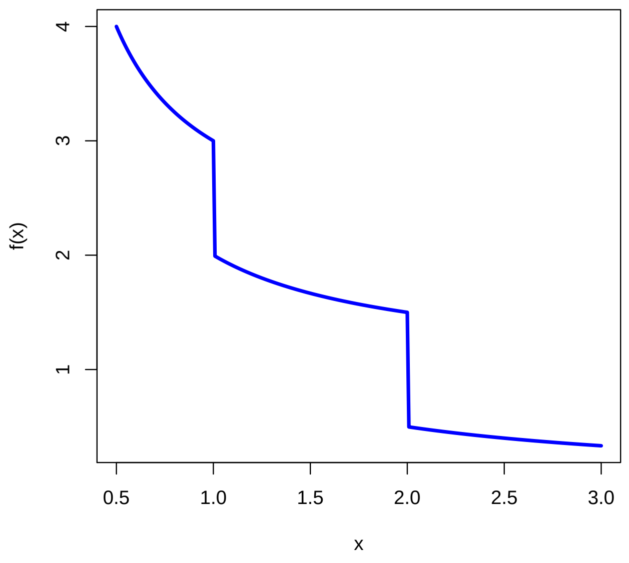

Figure 6: Three Section Plot

References on R

A long list of contributed documents on **`R`** which are available on the web can be found at
`http://cran.r-project.org/other-docs.html`
and links to document collections, Journals, and Proceedings can be found at
`http://www.r-project.org/other-docs.html`

A very useful 103 page tutor in both a pdf and html version is found with your local installation of **`R`**. On the author's computer it is located at **`C:\Program Files\R\R-3.0.1\doc\manual\R-intro.pdf`**
and **`C:\Program Files\R\R-3.0.1\doc\manual\R-intro.html`**, and is titled **"An Introduction to R: Notes on R: A Programming Environment for Data Analysis and Graphics"**, by W.N. Venables, D.M. Smith, and the R Core Team.

Whether or not you have some prior knowledge of **`Matlab`**, you will find a very well organized 52 page pdf comparison of **`Matlab`** and **`R`** syntax, organized by the type of operation desired, at
`http://www.math.umaine.edu/~hiebeler/comp/matlabR.html`.

An 83 page pdf introduction to **`R`** by active scientific researchers including the co-author of **"Solving Differential Equations in R"** is a pdf document titled **"Using R for Scientific Computing"** by Karline Soetaert and Filip Meysman(2011), which can be found at
`http://vkc.library.uu.nl/vkc/darwin/knowledgeportal/Lists/`
`Conferences/Attachments/28/Meysman_ScienceR.pdf`

and an earlier 46 page version dated 2008 with the single author Karline Soetaert:
`http://cran.r-project.org/doc/contrib/Soetaert_Scientificcomputing.zip`, which, when unzipped, contains ScientificComputing.pdf with document title "Using R for Scientific Computing".

The 2009 46 page version can be found at
`https://r-forge.r-project.org/scm/viewvc.php/pkg/marelacTeaching/inst/doc/lecture/?root=marelac&sortby=rev&pathrev=122`

1.3 Elements of the Maxima Language

Users new to Maxima should work through some of the tutorials which can be found under the Documentation tab on the Maxima CAS project page:
`http://maxima.sourceforge.net/` Chapters 1 - 3 of the author's **Maxima by Example** would be sufficient.

The author always uses the **`xmaxima.exe`** interface, located in **`...\bin`**. A useful keyboard feature of Xmaxima is the use of the two-key command **`Alt+p`**. Used once, or multiple times, you can easily repeat a previous command, but you first have the opportunity to edit it before pressing Enter.

The usual keyboard commands allow easy movement within the Xmaxima window: **`Home`** (beginning of line), **`End`** (end of line), **`PageUp`**, **`PageDown`**, **`UpArrow`**, **`DownArrow`**, **`Ctrl+Home`** (top of window), **`Ctrl+End`** (bottom of window).

See the discussion of how to arrange your work folder and the startup options in ch.1 of the author's set of notes: **`Maxima by Example`**. Google 'ted woollett' and those notes will be at the first hit. The author uses the startup file to send the command **`display2d:false`** to Maxima for routine use. This allows more information to be in the Xmaxima window at a time, and allows for easier selection and copying of portions of the work for transfer to a work text file or the verbatim section of a tex file. The code fragments copied into verbatim sections of a tex file also then take up less room when converted to a pdf file.

Case matters in Maxima.

```
(%i1) a : 2;
(%o1) 2
(%i2) A : 3;
(%o2) 3
(%i3) [a, A];
(%o3) [2,3]
(%i4) if a = A then print("equal") else print("not equal")$
not equal
```

Learn the crucial difference between using **`map`** and **`apply`** with a Maxima list. Suppose **`f`** is some core or homemade function (here the definition of **`f`** is unknown to Maxima):

```
(%i5) f;
(%o5) f
(%i6) map('f, [1,2,3]);
(%o6) [f(1),f(2),f(3)]
(%i7) apply('f,[1,2,3]);
(%o7) f(1,2,3)
(%i8) map('sin,[1.0, 2.0]);
(%o8) [0.84147,0.9093]
(%i9) apply('mod,[6,2]);
(%o9) 0
```

`mod` is the Maxima modulus function. A useful example of **`apply`** is with the **`"+"`** function:

```
(%i10) apply("+",[2,4,6,8]);
(%o10) 20
```

which adds up the elements in the given list. If you have a named list, the “apply” method is faster than using the core Maxima function **sum**:

```
(%i11) xv : makelist(i,i,1,10);
(%o11) [1,2,3,4,5,6,7,8,9,10]
(%i12) apply("+",xv);
(%o12) 55
(%i13) sum(xv[i],i,1,length(xv));
(%o13) 55
```

Many core Maxima functions (such as **float**) automatically distribute over lists, so the use of **map** is not always required to get the job done. The Lisp code which defines the core function determines whether or not that function distributes over lists. Let’s check the function **sin**:

```
(%i14) sin([1,2]);
(%o14) [sin(1),sin(2)]
(%i15) properties(sin);
(%o15) ["mirror symmetry",deftaylor,integral,"distributes over bags",rule,
        noun,gradef,transfun]
```

The Maxima function **properties** can be used with core functions and the element **"distributes over bags"** indicates that **sin** distributes over lists (provided a global parameter **distribute_over** is set to **true** (the default)). See the Maxima help manual (in XMaxima, select the menu item: Help, Maxima Manual), and using the **index** mode, type in the start of **distributes_over** and select help by pressing Enter. The result starts out:

```
Option variable: distribute_over
Default value: true

distribute_over controls the mapping of functions over bags
 like lists, matrices, and equations. At this time not all
 Maxima functions have this property. It is possible to
 look up this property with the command properties.

The mapping of functions is switched off, when setting
     distribute_over to the value false.

Examples:

The sin function maps over a list:

(%i1) sin([x,1,1.0]);
(%o1)                [sin(x), sin(1), .8414709848078965]
mod is a function with two arguments which maps over lists.
    Mapping over nested lists is possible too:

(%i2) mod([x,11,2*a],10);
(%o2)                   [mod(x, 10), 1, 2 mod(a, 5)]
(%i3) mod([[x,y,z],11,2*a],10);
(%o3)       [[mod(x, 10), mod(y, 10), mod(z, 10)], 1, 2 mod(a, 5)]
```

So let’s check that global Maxima parameter:

```
(%i16) distribute_over;
(%o16) true
(%i17) distribute_over:false$
(%i18) sin([1,2]);
(%o18) sin([1,2])
(%i19) distribute_over:true$
(%i20) sin([1,2]);
(%o20) [sin(1),sin(2)]
```

An example of a core Maxima function which does not distribute over lists is **print**:

```
(%i1) print([a,b,c]);
[a,b,c]
(%o1) [a,b,c]
(%i2) map('print,[a,b,c])$
a
b
c
(%i3) properties(print);
(%o3) [transfun]
```

Tests for equality involve **=**, tests for "not equal" involve **#**.

```
(%i4) if 1 = 2/2 then print("equal") else print("not equal")$
equal
(%i5) is(equal(1, 2/2));
(%o5) true
(%i6) if 3 # 4/2 then print("not equal") else print("equal")$
not equal
(%i7) is(equal(3, 4/2));
(%o7) false
```

A straightforward method to construct a list is to use the **makelist** function, which we have already used above. Here are some more simple examples:

```
(%i8) makelist(i/4,i,0,8);
(%o8) [0,1/4,1/2,3/4,1,5/4,3/2,7/4,2]
(%i9) makelist(i/4,i,0,8,2);
(%o9) [0,1/2,1,3/2,2]
(%i10) makelist(x^2,x,0,6);
(%o10) [0,1,4,9,16,25,36]
(%i11) makelist(x^2,x,0,9,3);
(%o11) [0,9,36,81]
```

The functions **rest**, **reverse**, **cons** and **length** are useful tools for working with Maxima lists:

```
(%i12) xv : makelist(i,i,1,10);
(%o12) [1,2,3,4,5,6,7,8,9,10]
(%i13) rest(xv);
(%o13) [2,3,4,5,6,7,8,9,10]
(%i14) rest(xv,2);
(%o14) [3,4,5,6,7,8,9,10]
(%i15) rest(xv,-1);
(%o15) [1,2,3,4,5,6,7,8,9]
(%i16) rest(xv,-2);
(%o16) [1,2,3,4,5,6,7,8]
(%i17) reverse(xv);
(%o17) [10,9,8,7,6,5,4,3,2,1]
(%i18) cons(0, xv);
(%o18) [0,1,2,3,4,5,6,7,8,9,10]
(%i19) xv;
(%o19) [1,2,3,4,5,6,7,8,9,10]
(%i20) first(xv);
(%o20) 1
(%i21) last(xv);
(%o21) 10
(%i22) xv[1];
(%o22) 1
(%i23) xv[10];
(%o23) 10
(%i24) length(xv);
(%o24) 10
```

A homemade Maxima function to make a table of values for a single known function is **ftable**:

```
ftable(func,x0,xf,dx):=
block([nL,fL,nfL,jj,ii],
   nL : makelist(zz,zz,x0,xf,dx),
   fL : map('func,nL),
   nfL : makelist(["  ",nL[jj],"  ",fL[jj]],jj,length(nL)),
   for ii thru length(nfL) do apply('print,nfL[ii]))$
```

After pasting this code into Xmaxima, one gets:

```
(%i1) ftable(sin,0,0.5,0.1);
   0    0
   0.1    0.099833416646828
   0.2    0.19866933079506
   0.3    0.29552020666134
   0.4    0.38941834230865
   0.5    0.4794255386042
(%o1) done
```

You can change the number of digits printed on the screen (the internal arithmetic is still 16 digit arithmetic) by using **fpprintprec**. You can also avoid the final **done** output by ending your input with the dollar **$** symbol instead of the semicolon **;**.

```
(%i2) fpprintprec:7$
(%i3) ftable(sin,0,0.5,0.1)$
   0    0
   0.1    0.099833
   0.2    0.19867
   0.3    0.29552
   0.4    0.38942
   0.5    0.47943
```

You can restore the default large number of digits printed to the screen using

```
(%i4) fpprintprec:0$
```

The construction of new Maxima functions is best done in a separate text editor with strong parenthesis and bracket matching behavior, such as **Notepad2** (excellent and free). When copying and pasting a piece of code into **Xmaxima**, such as

```
D1f(func,xval,[oa]) :=
block([h : 1e-8],
    if length(oa) > 0 then h : oa[1],
    (func(xval + h) - func(xval))/h)$
```

make sure your copy ends exactly with the dollar sign, and does not include extra white spaces after the dollar sign. The author finds that **Xmaxima** will "hang" and come to a halt if extra white space occurs after the dollar sign. In particular, pressing Enter after the paste causes nothing to happen. Then using **(File, Interrupt)** (equivalent to **Ctrl+g**) produces a Lisp error message, but the Maxima prompt does not reappear. One must use **(File, Restart)** to start up a fresh version of Maxima to get back a prompt.

1.3.1 The Maxima Function plot2d

The first curve example done using **R** can be done with one **plot2d** command using Maxima. If we use the default color cycle of Maxima, as in

```
plot2d([u^3,u^2,u],['u,-3,3],['y,-25,25],
        [style,[lines,3]],
        [gnuplot_preamble,"set grid;"])$
```

the default color scheme results in the first (cubic) function drawn in blue, the second (quadratic) function drawn in red, and the third (linear) function drawn in green.

To get the same color choices in our first **R** example, we need a more involved **plot2d** command, which forces color choices for each of the three functions in the list:

```
plot2d([u^3,u^2,u],['u,-3,3],['y,-25,25],
        [style,[lines,3,3],[lines,3,1],[lines,3,2]],
        [gnuplot_preamble,"set grid;"])$
```

The **line** syntax is **[lines, width, color]**, with the default line width being **1**, and the color choices being: 1:blue, 2:red, 3: green, 4: violet, 5: brown, 6:black, 7:blue, ...

To get the same color choices in our second **R** example, we can use

```
plot2d([u^5,u^3,u^2-5],['u,-5,5],['y,-10,25],
        [style,[lines,3,2],[lines,3,6],[lines,3,1]],
        [gnuplot_preamble,"set grid;"])$
```

Much more information about Maxima's **plot2d** function can be found in Chapter 2 of **Maxima by Example** on the author's webpage.

1.4 Numerical Derivatives

See **http://en.wikipedia.org/wiki/Numerical_derivative** for a discussion of different types of numerical derivative approximations.

In Computational Physics, ch 1, Sec.1, Koonin uses Taylor series expansions to derive what he calls a "3 point" approximation to the first derivative of a function **f** at some point **x**

```
   f'(x) = ( f(x+h) - f(x-h) )/(2*h)  -(h^2/6)*f'''(x) + O(h^4)
```

The second and third terms on the right hand side (the error terms) are zero if the third and higher derivatives of the given function vanish at the position **x**, which would be the case if the function is an arbitrary second degree polynomial in the interval **[x-h, x+h]**. (All the error terms in the above involve even powers of **h**.) If the function's third derivative is not zero, the error terms can in principle be made arbitrarily small by letting **h** shrink to zero. However arithmetic in **R** is done with only 16 digit accuracy, and this means there is a limit to how small we can take **h** in practice, related to the process of finding the difference of two almost equal floating point numbers and in the process losing significant digits of precision (roundoff error).

We will call the following a "central difference" approximation to the numerical derivative of **f** at **x**

```
   f'(x) = ( f(x+h) - f(x-h) )/(2*h) + O(h^2)
```

For this central difference approximation, and in the absence of roundoff error, we expect the errors to decrease by roughly a factor of 100 if **h** is decreased by a factor of 10.

We call the following a "forward difference" approximation:

```
   f’(x) = ( f(x+h) - f(x) )/h  + O(h)
```

and call the following a "backward difference" approximation:

```
   f’(x) = ( f(x) - f(x-h) )/h  + O(h)
```

Koonin shows that the latter two approximations have error terms **O(h)** and thus the errors are expected to decrease by roughly a factor of 10 if we decrease the value of **h** by a factor of 10 (in the absence of roundoff error).

A difference formula for the 3 point symmetric **second** derivative is

```
   f’’(x) = ( f(x+h) -2*f(x) + f(x-h) )/h^2 + O(h^2)
```

1.4.1 Numerical Derivative Functions in R

Numerical Derivative Functions in the Package rootSolve

The package **rootSolve** has the functions **gradient** and **hessian** which return matrices containing the first and second derivatives of a function, using the simple difference formulas, which have, in general, less accuracy than the functions provided by the package **numDeriv**, to be discussed later.

By default, **rootSolve::gradient** and **rootSolve::hessian** use the forward difference method, but by including an option arg they can be required to use the centered difference method. After loading the package **rootSolve** using the **library** function, you can use **?gradient** and **?hessian** to see documentation.

rootSolve::gradient

Part of the **gradient** documentation is:

```
gradient(f, x, centered = FALSE, pert = 1e-8, ...)
Arguments:
f: function returning one function value, or a vector of function values.
x: either one value or a vector containing the x-value(s) at which
    the gradient matrix should be estimated.
centered: if TRUE, uses a centered difference approximation,
  else a forward difference approximation.
pert: numerical perturbation factor; increase depending on precision
   of model solution.
... : other arguments passed to function f
```

Note that the step size arg can be in any of the forms: **pert = 1e-4**, **pert = 0.0001**, or **pert = 10^(-4)**.

We test **gradient** with the first derivative of **sin(x)** at **x = 1**. Because **gradient** returns a matrix as its value, we extract the single matrix element returned using the bracket syntax **[1,1]** (row 1, column 1).

```
> library(rootSolve)
> ?gradient
> options(digits=16)
> exact = cos(1); exact
[1] 0.5403023058681398
```

```
> numd1 = gradient(sin, 1)[1,1]; numd1
[1] 0.5403023028982545
> (numd1 - exact)/exact
[1] -5.496710257161753e-09
> options(digits=3)
> hv = c(6,8,10)
> for (i in 1:3) {
+     n = hv[i]
+     num.d = gradient(sin, 1, pert = 10^(-n))[1,1]
+     cat(" h = ",10^(-n)," error = ", (num.d - exact)/exact,"\n")}
 h =  1e-06  error =  -7.79e-07
 h =  1e-08  error =  -5.5e-09
 h =  1e-10  error =  -1.08e-07
```

We see the effects of roundoff error if too small a step-size is used (subtraction of almost equal numbers leading to loss of digits of precision).

In the above test, we used the default forward difference approximation. Let's try the **centered difference approximation** next.

```
> options(digits=16)
> numd3 = gradient(sin, 1, centered=TRUE)[1,1]; numd3
[1] 0.5403023084493697
> (numd3 - exact)/exact
[1] 4.777380863375732e-09
> options(digits=3)
> for (i in 1:3) {
+     n = hv[i]
+     num.d = gradient(sin, 1, centered=TRUE, pert = 10^(-n))[1,1]
+     cat(" h = ",10^(-n)," error = ", (num.d - exact)/exact,"\n")}
 h =  1e-06  error =  5.13e-11
 h =  1e-08  error =  4.78e-09
 h =  1e-10  error =  -1.08e-07
```

We see that the step size `1e-6` (smaller than the default), when used with the centered method, leads to much higher accuracy.

Let's make a table, using `data.frame`, of fractional errors which compares the forward with the centered method for various values of `h`. The vector of `h` values (the step size) and the fractional error calculations make use of `R`'s vector arithmetic abilities.

```
> options(digits=3)
> hv = 4:10 ; hv
[1]  4  5  6  7  8  9 10
> numd = vector(mode="numeric",length=length(hv))
> for (i in 1:length(hv)){
+       numd[i] = gradient(sin,1,pert = 10^(-hv[i]))[1,1]}
> numd.c = vector(mode="numeric",length=length(hv))
> for (i in 1:length(hv)){
+       numd.c[i] = gradient(sin,1,pert = 10^(-hv[i]), centered=TRUE)[1,1]}
> data.frame( h = 10^(-hv),  forward = (exact - numd)/exact ,
+                centered = (exact - numd.c)/exact )
      h   forward  centered
1 1e-04  7.79e-05  1.67e-09
2 1e-05  7.79e-06  2.06e-11
3 1e-06  7.79e-07 -5.13e-11
4 1e-07  7.74e-08  3.60e-10
5 1e-08  5.50e-09 -4.78e-09
6 1e-09 -9.72e-08  5.50e-09
7 1e-10  1.08e-07  1.08e-07
```

rootSolve::hessian

Part of the `rootSolve::hessian` documentation is

```
hessian(f, x, centered = FALSE, pert = 1e-8, ...)
Arguments:
f: function returning one function value, or a vector of function values.
x: either one value or a vector containing the x-value(s) at
   which the hessian matrix should be estimated.
centered: if TRUE, uses a centered difference approximation,
      else a forward difference approximation.
pert: numerical perturbation factor; increase depending on
         precision of model solution.
... : other arguments passed to function f.
Details:
Function hessian(f,x) returns a forward or centered difference
  approximation of the gradient, which itself is also
  estimated by differencing. Because of that, it is not very precise.
```

We test this function by calculating the second derivative of `sin(x)` at the point `x = 1`.

```
> ?hessian
> options(digits=16)
> exact = -sin(1); exact
[1] -0.8414709848078965
> numd1 = hessian(sin, 1)[1,1]; numd1
[1] -0.5403023028982545
> (numd1 - exact)/exact
[1] -0.3579073875950663
> numd2 = hessian(sin, 1,centered=TRUE)[1,1]; numd2
[1] -0.8252626693128207
> (numd2 - exact)/exact
[1] -0.01926188280725582
```

Let’s concentrate on the centered method which so far seems more accurate.

```
> hv = 2:7; hv
[1] 2 3 4 5 6 7
> numd.c = vector(mode="numeric",length=length(hv))
> for (i in 1:length(hv)){
+       numd.c[i] = hessian(sin, 1, pert = 10^(-hv[i]), centered=TRUE)[1,1]}
> options(digits=3)
> data.frame( h = 10^(-hv), centered = (exact - numd.c)/exact )
      h  centered
1 1e-02  1.64e-05
2 1e-03 -2.32e-06
3 1e-04 -1.71e-05
4 1e-05  1.36e-04
5 1e-06 -3.83e-03
```

So for this function `sin`, the most accurate second derivative (using the centered method) produced by `rootSolve::hessian` occurs for (approx) `pert=1e-3`. The results produced by `numDeriv::hessian` are much more accurate.

Numerical Derivative Functions in the Package numDeriv

More accurate derivatives can be calculated using the package **numDeriv**, which includes the function `grad` which can be used for calculation of the first derivative and the function `hessian` which can be used to calculate the second derivative.

Because both of the packages **rootSolve** and **numDeriv** define a function with the name ‘hessian”, it is simplest to start up a fresh instance of **R** in which the package **rootSolve** has not been loaded (rather than trying to use `detach` and investigating the validity of the consequent results).

numDeriv::grad

Part of the documentation produced by using **`?grad`** after loading the **numDeriv** package using **`library`** is:

```
grad(func, x, method="Richardson", method.args=list(), ...)
Arguments:
func: a function with a scalar real result (see details).
x: a real scalar or vector argument to func, indicating the
   point(s) at which the gradient is to be calculated.
method: one of "Richardson", "simple", or "complex" indicating
         the method to use for the approximation.
method.args: arguments passed to method. Arguments not specified
   remain with their default values as specified in details
... :  additional arguments passed to func. WARNING: None of
    these should have names matching other arguments of this function.
Value: A real scalar or vector of the approximated gradient(s).
```

We use **`grad`** to calculate the first derivative of $\sin(x)$ at the point $x = 1$. For our example, **`grad`** returns a number (not a matrix).

```
> library(numDeriv)
> ?grad
> options(digits=16)
> exact = cos(1); exact
[1] 0.5403023058681398
> numd1 = grad(sin, 1); numd1
[1] 0.5403023058635031
> (numd1 - exact)/exact
[1] -8.581537349340139e-12
> numd2 = grad(sin, 1,method = "complex"); numd2
[1] 0.5403023058681398
> (numd2 - exact)/exact
[1] 0
> numd3 = grad(sin, 1,method = "simple"); numd3
[1] 0.5402602314186211
> (numd3 - exact)/exact
[1] -7.787205248188559e-05
```

The default method is **`"Richardson"`**, which uses Richardson extrapolation (see the **`?grad`** help documentation for details and references). The **`"complex"`** method assumes the given function is analytic in a neighborhood of the evaluation point **`x`**.

numDeriv::hessian

Part of the documentation for **`hessian`** is

```
hessian(func, x, method="Richardson", method.args=list(), ...)
Arguments:
func: a function for which the first (vector) argument is used as a parameter vector.
x: the parameter vector first argument to func.
method: one of "Richardson" or "complex" indicating the method to
     use for the approximation.
method.args: arguments passed to method. See grad.
        (Arguments not specified remain with their default values.)
... : an additional arguments passed to func. WARNING: None of
      these should have names matching other arguments of this function.
Value: An n by n matrix of the Hessian of the function calculated
    at the point x.
```

We use **numDeriv::hessian** to calculate the second derivative of $\sin(x)$ at the point $x = 1$. We need to extract the single matrix element returned for our example using **[1,1]** (row 1, column 1).

```
> ?hessian
> exact = -sin(1); exact
[1] -0.8414709848078965
> numd1 = hessian(sin, 1)[1,1]; numd1
[1] -0.841470984807975
> (numd1 - exact)/exact
[1] 9.328042114122101e-14
> numd2 = hessian(sin, 1, method = "complex")[1,1]; numd2
[1] -0.8414709848078918
> (numd2 - exact)/exact
[1] -5.541411156904218e-15
```

The **"simple"** method is not supported by **numDeriv::hessian**.

1.4.2 Testing Simple Numerical Derivative Methods in R

To test the symmetric centered difference method of finding an approximate numerical derivative, Koonin presents a short interactive program in which the user provides a value of step size **h** and the central difference approximation to the first derivative of **sin(x)** is computed at **x = 1**, together with the error. The "exact" answer is **cos(1)**. The default number of digits displayed in **R** is **7**, which can be changed by the user via **options(digits = n)**. (Use **options()** to see all current options settings, but beware because there are many options.)

```
> cos(1)
[1] 0.5403023
> options(digits=16)
> cos(1)
[1] 0.5403023058681398
> options(digits=7)
```

Here is **R** code which mimics his code:

```
tryh = function() {
     x = 1
     exact = cos(x)
     cat(" enter h <= 0 to stop \n")
     repeat {
       h = as.numeric(readline(" input h: "))
       if (h <= 0) break
       fprime = 0.5*(sin(x+h) - sin(x-h))/h
       diff = exact - fprime
       cat(" h = ",h," error = ",diff," \n\n")}}
```

After pasting this code into the **R** Console window, we get

```
> tryh()
 enter h <= 0 to stop
 input h: 0.5
 h =  0.5  error =  0.02223286
 input h: 1e-6
 h =  1e-06  error =  -2.771694e-11
 input h: 1e-8
 h =  1e-08  error =  -2.58123e-09
 input h: 1e-7
 h =  1e-07  error =  1.943277e-10
 input h: -1
 >
```

We see that roundoff error occurs for **`h < 1e-6`** with this central difference method and with this function.

We have used the **`R`** function **`repeat()`** to define an "endless loop" and have used the **`break`** function to get out of the loop.

There are at least two ways to control the number of digits printed in the output. The first (and easiest) is to use **`options(digits = 3)`**, say.

```
> options(digits = 3)
```

```
> tryh()
 enter h <= 0 to stop
 input h: 1e-5
 h =  1e-05  error =  1.11e-11
 input h: 1e-6
 h =  1e-06  error =  -2.77e-11
 input h: -1
 >
```

The second is to replace the last line in tryh() with the two lines:

```
         str1 = sprintf(" h = %.2e   error = %.2e", h, diff)
         cat(paste(str1, "\n\n") )  }}
```

in which **`sprintf`** allows the use of a formatting string as in the **`C`** language.

In order to make a table comparing the errors in evaluating the first derivative of **`sin(x)`** at the point **`x = 1`**, we define the functions **`D1c`**, **`D1f`**, and **`D1b`** which employ the central, forward, and backward difference formulas, respectively.

```
D1c = function (func, x, h = 1e-8) {
        (func(x + h) - func(x - h))/(2*h) }
D1f = function(func,x,h = 1e-8) {
        (func(x + h) - func(x))/h }
D1b = function(func,x,h = 1e-8) {
        (func(x) - func(x - h))/h }
```

The "default" value of **`h`**, if the third slot is not used, is set to be **`1e-8`**. With these functions pasted into **`R`**, for example, we have

```
> exact = cos(1)
> exact - D1c(sin,1)
[1] -2.58e-09
> exact - D1c(sin,1,1e-6)
[1] -2.77e-11
> exact - D1c(sin,1, c(1e-5, 1e-6))
[1]  1.11e-11 -2.77e-11
```

In the last entry, instead of a single value of **`h`**, we have used a vector of **`h`** values, in the form **`c(h1,h2)`** and obtained the error for two cases using only one line.

Using this approach we now construct a table of errors (with headings), using the central, forward, and backward difference methods, in computing the first derivative of **`sin(x)`** at **`x = 1`**.

```
> hv = 10^(-(4:9)); hv
[1] 1e-04 1e-05 1e-06 1e-07 1e-08 1e-09
> merror = function(fun) exact - fun(sin, 1, hv)
> merror(D1c)
[1]  9.004295087322589e-10  1.114086600750852e-11 -2.771693985437196e-11
[4]  1.943276650706594e-10 -2.581229896492232e-09  2.969885226633551e-09
> data.frame(h=hv,D1c=merror(D1c),D1f=merror(D1f),D1b=merror(D1b))
      h       D1c       D1f       D1b
1 1e-04  9.00e-10  4.21e-05 -4.21e-05
2 1e-05  1.11e-11  4.21e-06 -4.21e-06
3 1e-06 -2.77e-11  4.21e-07 -4.21e-07
4 1e-07  1.94e-10  4.18e-08 -4.14e-08
5 1e-08 -2.58e-09  2.97e-09 -8.13e-09
6 1e-09  2.97e-09 -5.25e-08  5.85e-08
```

We see that the error of the derivative approximations cannot be made arbitrarily small by continuing to decrease the size of `h`, due the the appearance of roundoff errors made in subtracting two almost equal numbers from each other using 16 digit arithmetic.
The **R** function `diff` is very efficient in taking numerical differences. The difference `b - a = diff(c(a, b))`.

```
> diff(c(3.1, 4.2))
[1] 1.1
```

We can improve `D1c`, for example, by employing the `diff()` function, but the resulting function will not work in the same way (as above) if the parameter `h` is set equal to a vector. Here we use `diff` to explore the roundoff error when using the central difference formula with `h = 1e-14`.

```
> options(digits = 16)
> h = 1e-14
> fp = sin(1 + h); fp
[1] 0.8414709848079019
> fm = sin(1-h); fm
[1] 0.8414709848078911
> df = diff(c(fm,fp)); df
[1] 1.088018564132653e-14
> df = df/(2*h); df
[1] 0.5440092820663267
> diff(c(df, cos(1)))
[1] -0.00370697619818694
> cos(1)
[1] 0.5403023058681398
```

For this small value of `h` the numerical derivative is returned with only two digits of accuracy.

1.4.3 Testing Simple Numerical Derivative Methods in Maxima

In Computational Physics, ch 1, Sec.1, Koonin has a short interactive program in which the user provides a value of `h` and the central difference approximation to the first derivative of `sin(x)` is computed at `x = 1`, together with the error. The symbolic answer is `cos(1)`, which is, to 16 digit accuracy,

```
(%i1) float(cos(1));
(%o1) 0.54030230586814
```

In this section on simple numerical derivatives in Maxima, we will not be as careful with error calculations as we will be in other sections. A more careful approach would be to define an “exact” value, good to 20 digits using bigfloat methods, as in

```
(%i2) exact : block([fpprec:20], bfloat(cos(1)));
(%o2) 5.403023058681397174b-1
```

and then compare a result **val**, calculated with some other method using 16 digit arithmetic, with the "exact value" via

```
  block([fpprec:20], bfloat(val - exact))
```

for "absolute error", or

```
  block([fpprec:20], bfloat( (val - exact)/exact))
```

for "fractional error".

If we translate the **R** version of **tryh()** (above) into Maxima, we must end each command with a comma, we must use **:** for assignment instead of **=**, we must replace curly brackets with parentheses, we must use **read** or **readonly** instead of **readline**, we must use **do** instead of **repeat** to get an endless loop, we must use **return** instead of **break** to get out of the loop, we must use **print** instead of **cat**, the **if** syntax for Maxima requires a **then**, we can add **numer:true** to ensure trig functions are converted to floating point numbers (or else wrap the output in **float**), and, finally, we should avoid the Maxima reserved word **diff** (used for symbolic differentiation in Maxima) in our code.

The difference between the Maxima functions **read** and **readonly** is that **read** evaluates the input and **readonly** does not evaluate the input.

```
tryh() :=
block([x:1, exact,h,fprime,fp_diff],numer:true,
     exact : cos(x),
     print(" enter h <= 0 to stop "),
     do (
       h : read(" input h: "),
       if h <= 0 then return(done),
       fprime : 0.5*(sin(x+h) - sin(x-h))/h,
       fp_diff : exact - fprime,
       print(" h = ",h," error = ",fp_diff )))$
```

After pasting this definition of **tryh()** into Maxima, we get

```
(%i3) fpprintprec:4$
(%i4) tryh();
 enter h <= 0 to stop
 input h:
1/20;
 h =  0.05  error =  2.251E-4
 input h:
1/40;
 h =  0.025  error =  5.628E-5
 input h:
1e-3;
 h =  0.001  error =  9.005E-8
 input h:
-1;
(%o4) done
```

In order to make a table comparing the errors in evaluating the first derivative of **sin(x)** at the point **x = 1**, we define the functions **D1c**, **D1f**, and **D1b** which employ the central, forward, and backward difference formulas, respectively.

In **R** we had the definition, incorporating a default value of **h** if **D1c** was called with only the first two args supplied:

```
D1c = function (func, x, h = 1e-8) {
          (func(x + h) - func(x - h))/(2*h) }
```

To construct a similar (in result) function in Maxima takes a little more work. First of all, list arithmetic in Maxima :

```
(%i5) 1 + [0.1, 0.01];
(%o5) [1.1,1.01]
(%i6) 1/[2,3];
(%o6) [1/2,1/3]
```

is similar to list arithmetic in **R**.

```
> 1 + c(0.1,0.01)
[1] 1.10 1.01
> 1/c(2,3)
[1] 0.5000000 0.3333333
```

However, to incorporate the default value for **h**, using Maxima, we need to use a special syntax,
D1c(func,xval,[oa]) := etc, etc., which we explain here.

In this Maxima function definition, inside the code, **oa** (other args) will appear as a list, and the list **oa** (inside the function) will be the zero length list **[]** if no third arg is supplied, will be a list containing one number **[h1]** if a number is supplied for the third arg, and will be a list of a list of **h** values if a list is supplied for the third arg. Here is a little test function to explore this behavior:

```
(%i7) test(func,xval,[oa]) := (print(" oa = ",oa))$
(%i8) test(sin,1)$
 oa =  []
(%i9) test(sin,1,1e-4)$
 oa =  [1.0E-4]
(%i10) test(sin,1,[1e-2,1e-4])$
 oa =  [[0.01,1.0E-4]]
```

The simplest version of Maxima code results from the assumption that the function used in the first slot (**func**) distributes over lists in the same way the core function **sin** does. We assume, in particular, that homemade functions used with **D1c** distribute over lists, in the same way as in these four examples:

```
(%i11)  ff(x) := x^2$
(%i12) ff([1,2,3]);
(%o12) [1,4,9]
(%i13) gg(x) := x^2*sin(x)$
(%i14) gg([1,2,3]);
(%o14) [sin(1),4*sin(2),9*sin(3)]
(%i15) kk(x) := sin(x)/x$
(%i16) kk([1,2,3]);
(%o16) [sin(1),sin(2)/2,sin(3)/3]
(%i17) jj(x) := x^2/cos(x)$
(%i18) jj([1,2,3]);
(%o18) [1/cos(1),4/cos(2),9/cos(3)]
```

We can then avoid the need to use **map** to get the function to act on each member of a list, and use the following simple code (which will do the job in all three cases):

```
D1c(func,xval,[oa]) :=
block([h : 1e-8],numer:true,
    if length(oa) > 0 then h : oa[1],
    (func(xval + h) - func(xval - h))/(2*h))$
```

We see the default value of **h** defined in the local variable list at the start of the **block** expression. We could just as well have used the syntax **block([h], h:1e-8, etc.** to define the default value of **h** in the code.

For a function which does **not** distribute over a list, one could use the code (which uses the Maxima function `listp`):

```
D1c_b(func,xval,[oa]) :=
block([h : 1e-8],numer:true,
   if length(oa) > 0 then h : oa[1],
   if listp(h) then
          (map(func,xval + h) - map(func,xval - h))/(2*h)
   else (func(xval + h) - func(xval - h))/(2*h))$
```

Here is practice using `D1c`:

```
(%i1) D1c(func,xval,[oa]) :=
block([h : 1e-8],numer:true,
    if length(oa) > 0 then h : oa[1],
    (func(xval + h) - func(xval - h))/(2*h))$
(%i2) fpprintprec:8$
(%i3) exact : float(cos(1));
(%o3) 0.540302
(%i4) exact - D1c(sin,1);
(%o4) -2.5812299E-9
(%i5) exact - D1c(sin,1,1e-4);
(%o5) 9.00429509E-10
(%i6) exact - D1c(sin,1,[1e-4,1e-5]);
(%o6) [9.00429509E-10,1.1140977E-11]
```

With the same assumptions, the forward and backward approximations to the first derivative are defined as:

```
D1f(func,xval,[oa]) :=
block([h : 1e-8],numer:true,
    if length(oa) > 0 then h : oa[1],
    (func(xval + h) - func(xval))/h)$

D1b(func,xval,[oa]) :=
block([h : 1e-8],numer:true,
    if length(oa) > 0 then h : oa[1],
    (func(xval) - func(xval - h))/h)$
```

We then can construct a table of errors vs. the size of `h` *after(!)* pasting in the definitions of `D1c`, `D1f`, and `D1b`.

```
(%i7) hv : makelist(10^(-n),n,4,9)$
(%i8) fpprintprec:4$
(%i9) float(hv);
(%o9) [1.0E-4,1.0E-5,1.0E-6,1.0E-7,1.0E-8,1.0E-9]
(%i10) merror(fun) := exact - fun(sin,1,hv)$
(%i11) merror(D1c);
(%o11) [9.0043E-10,1.1141E-11,-2.7717E-11,1.9433E-10,-2.5812E-9,2.9699E-9]
(%i12) fpprintprec:2$
(%i13) (print("  h  ","  D1c   ","   D1f   ","   D1b   "),
  for i thru length(hv) do
     print(float(hv[i]), merror(D1c)[i], merror(D1f)[i],
                    merror(D1b)[i]))$
  h     D1c       D1f       D1b
1.0E-4 9.0E-10 4.21E-5 -4.21E-5
1.0E-5 1.11E-11 4.21E-6 -4.21E-6
1.0E-6 -2.77E-11 4.21E-7 -4.21E-7
1.0E-7 1.94E-10 4.18E-8 -4.14E-8
1.0E-8 -2.58E-9 2.97E-9 -8.13E-9
1.0E-9 2.97E-9 -5.25E-8 5.85E-8
```

We see again that the error of the derivative approximations cannot be made arbitrarily small by continuing to decrease the size of `h`, due the the appearance of roundoff errors made in subtracting two almost equal numbers from each other using 16 digit arithmetic.

1.5 Numerical Quadrature

In Computational Physics, Ch.1, Sec.2, Koonin discusses the trapezoidal rule for a uniform grid, two versions (**1/3, 3/8**) of Simpson's rule for a uniform grid, and Bode's rule.

When we use **R** or Maxima to solve complicated physics problems, we will choose the most accurate and efficient means available. If the software available does not have what is needed or what works, homemade functions will be resorted to. The homemade examples for quadrature (trapezoidal and Simpson's rules) presented here will not necessarily be used in our physics explorations, but understanding how they work will introduce you to the **R** and Maxima syntax for getting things done. We will check the accuracy of these very simple quadrature rules using the quadrature functions available in **R** and Maxima.

1.5.1 R function integrate for one dimensional integrals

The **R** language software has the numerical integration function **integrate** with the syntax

```
    integrate(fun, a, b, ...)
```

Use of **?integrate** brings up complete syntax and return named values information. For example, the returned named values info is:

```
Value
A list of class "integrate" with components

1. value
the final estimate of the integral.
2. abs.error
estimate of the modulus of the absolute error.
3. subdivisions
the number of subintervals produced in the subdivision process.
  the default is 100L
4. message
"OK" or a character string giving the error message.
5. call
the matched call.
```

One can use abbreviations for the names of these returned values. Here are examples of quadrature over a finite interval.

```
> integrate(sin,0,1)
0.4596977 with absolute error < 5.1e-15
> fun = function(u) u*sin(u)^2
> integrate(fun, 0, 1)
0.199694 with absolute error < 2.2e-15
> integrate(function(x) sqrt(x)*log(1/x),0,1)
0.4444444 with absolute error < 1.3e-07
> integrate(function(x) sqrt(x)*log(1/x),0,1,rel.tol = 1e-10)
0.4444444 with absolute error < 4.9e-16
> ival = integrate(function(x) sqrt(x)*log(1/x),0,1,rel.tol = 1e-10)
> cat(ival$value, ival$abs.error, ival$subdivisions, ival$message)
0.4444444 4.934325e-16 8 OK
> cat(ival$val, ival$abs.e, ival$sub, ival$mes)
0.4444444 4.934325e-16 8 OK
> cat(ival$v, ival$a, ival$s, ival$m)
0.4444444 4.934325e-16 8 OK
```

Here is an example of the **R** function **integrate** used for an unbounded interval quadrature.

```
> ival = integrate(function(x) x^2*exp(-4*x),0, Inf, rel.tol = 1e-8)
> cat(ival$value, ival$abs.error)
0.03125 2.959161e-11
> cat(ival$v, ival$a)
0.03125 2.959161e-11
```

Here is an example in which the number of subdivisions needs to be increased from the default value of `100L`.

```
> integrate(function(x) sin(x)/(1+x^2),0 , Inf)
Error in integrate(function(x) sin(x)/(1 + x^2), 0, Inf) :
  maximum number of subdivisions reached
> integrate(function(x) sin(x)/(1+x^2),0 ,Inf,subdivisions=300L)
0.6467757 with absolute error < 9.1e-05
```

1.5.2 R function elliptic::myintegrate for integration of a complex function

`R`'s function `integrate` will only accept a real function.

```
> integrate(function(x) 1i*sin(x),0,1)
Error in integrate(function(x) (0+1i) * sin(x), 0, 1) :
  evaluation of function gave a result of wrong type
```

The package `elliptic` contains the functions `myintegrate`, `integrate.contour`, and `integrate.segments`. The latter two functions allow numerical integration in the complex plane.

Here is a simple example of using `myintegrate`. We know that $\mathrm{e}^{\mathrm{i}\,\mathrm{x}} = \cos(\mathrm{x}) + \mathrm{i}\sin(\mathrm{x})$, so $\int_0^1 \mathrm{e}^{\mathrm{i}\,\mathrm{x}}\,\mathrm{dx} = \int_0^1 \cos(\mathrm{x})\,\mathrm{dx} + \mathrm{i}\,\int_0^1 \sin(\mathrm{x})\,\mathrm{dx}$.

`R` does not recognize the `i` in the expression `i*sin(1)`, but does recognize `1i*sin(1)`.

```
> cos(1) + i*sin(1)
Error: object 'i' not found
> cos(1) + 1i*sin(1)
[1] 0.5403023+0.841471i
```

After installing the package `elliptic`,

```
> integrate(function(x) cos(x),0,1)$val +
+           1i*integrate(function(x) sin(x),0,1)$val
[1] 0.841471+0.4596977i
> library(elliptic)
> myintegrate(function(x) exp(1i*x),0,1)
[1] 0.841471+0.4596977i
```

1.5.3 R function cubature::adaptIntegrate for multi-dimensional quadrature

This function only accepts integration over a finite domain, and does not allow for variable upper limits on inner integrals. See `http://ab-initio.mit.edu/wiki/index.php/Cubature` for some useful background to the `cubature` package, including the quote:

> This algorithm is best suited for a moderate number of dimensions (say, `< 7`), and is superseded for high-dimensional integrals by other methods (e.g. Monte Carlo variants or sparse grids).

To use **adaptIntegrate** for double or higher multi-dimension integrals, you first need to download the package **cubature**. If you are using **RStudio**, go to the packages panel and click on 'Install Packages...'.

```
> install.packages("cubature")
trying URL 'http://cran.rstudio.com/bin/windows/contrib/3.0/cubature_1.1-2.zip'
Content type 'application/zip' length 47448 bytes (46 Kb)
opened URL
downloaded 46 Kb

package cubature successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Documents and Settings\Edwin Woollett\Local Settings\Temp\RtmpGA2HLF\downloaded_packages
```

You then need to use **library** to load the package into your current work session. You can then use **?name** to get the documentation and description of required and optional arguments, and (named) return value(s).

```
> library(cubature)
> ?adaptIntegrate
```

Part of the documentation is:

```
adaptIntegrate {cubature}
Adaptive multivariate integration over hypercubes

Description:
The function performs adaptive multidimensional integration
  (cubature) of (possibly) vector-valued integrands over hypercubes.

Usage:
adaptIntegrate(f, lowerLimit, upperLimit, ...,
      tol = 1e-05, fDim = 1, maxEval = 0,
          absError=0, doChecking=FALSE)

Arguments:

f
The function (integrand) to be integrated
lowerLimit
The lower limit of integration, a vector for hypercubes
upperLimit
The upper limit of integration, a vector for hypercubes
...
All other arguments passed to the function f
tol
The maximum tolerance, default 1e-5.
fDim
The dimension of the integrand, default 1, bears no
             relation to the dimension of the hypercube
maxEval
The maximum number of function evaluations needed,
    default 0 implying no limit
absError
The maximum absolute error tolerated
doChecking
A flag to be a bit anal about checking inputs to
   C routines. A FALSE value results in approximately
   9 percent speed gain in our experiments. Your
   mileage will of course vary. Default value is FALSE.
```

```
Value
The returned value is a list of four items:

1. integral
the value of the integral
2. error
the estimated relative error
3. functionEvaluations
the number of times the function was evaluated
4. returnCode
the actual integer return code of the C routine
```

A return code of **`0`** means no problems were encountered. The lowerLimit arg is a single number for a one dimensional integral, and is a vector (for example, **`c(-1, 4)`** for a two dimensional integral), likewise for the upperLimit arg. See the examples below.

We first use a one dimensional test integral chosen from the documentation examples, first getting the exact value of the integral using Maxima

```
(%i7) integrate(sin(4*x)*x*((x*(x*(x*x-4) + 1) - 1)), x, -2, 2);
(%o7) (703*sin(8)+536*cos(8))/512+(447*sin(8)+1528*cos(8))/512
(%i8) float(%);
(%o8) 1.635644362960763
```

and then using **`adaptIntegrate`** in **`R`**:

```
> int1d = function(x) sin(4*x)*x*((x*(x*(x*x-4) + 1) - 1))
> intval = adaptIntegrate(int1d, -2, 2, tol=1e-7)
> cat(intval$i, intval$e)
1.635644 4.024021e-09
> cat(intval$f, intval$r)
105 0
```

As a two dimensional test integral we use Maxima to find the exact value of the integral $\int_0^1 \left(\int_2^3 \mathbf{y}\, \cos(\mathbf{x} + \mathbf{y})\, \mathbf{dx} \right) \mathbf{dy}$.

```
(%i5) integrate(integrate(y*cos(x+y),x,2,3),y,0,1);
(%o5) sin(4)-cos(4)-2*sin(3)+cos(3)+sin(2)
(%i6) float(%);
(%o6) -0.46609396033881
```

An anonymous function method of using **`adaptIntegrate`** is, with **`c(2,0)`** being the lower limits (**`2`** for the lower limit of the first integral done over the real variable **`x`**, **`0`** for the lower limit of the second integral done over the real variable **`y`**) and **`c(3,1)`** being the upper limits:

```
> adaptIntegrate(function(z) z[2]*cos(z[1] + z[2]), c(2,0), c(3,1))
$integral
[1] -0.466094
$error
[1] 1.66805e-06
$functionEvaluations
[1] 17
$returnCode
[1] 0
```

Or, abbreviating ’integral’

```
> adaptIntegrate(function(z) z[2]*cos(z[1] + z[2]), c(2,0), c(3,1))$int
[1] -0.466094
```

To use a named function we can define

```
fun = function(z) {
   x1 = z[1]
   x2 = z[2]
   x2*cos(x1 + x2)}
```

whose use produces;

```
> adaptIntegrate(fun, c(2,0), c(3,1))$int
[1] -0.466094
```

1.5.4 Maxima quadrature functions quad_qags and quad_qagi

Maxima has various methods for quadrature (see Ch. 8 and 9 of **Maxima by Example** on the author’s webpage).

The Maxima functions **`quad_qags`** (for a finite interval) and **`quad_qagi`** (for unbounded intervals) require integrands which evaluate to real floating point numbers. We discuss their use with complex integrands below.
Here is an example of using **`quad_qags`** for **finite interval** quadrature, taken from the Maxima help manual entry for **`quad_qags`**, followed by use of Maxima’s **`integrate`** function which tries to find an exact symbolic value (instead of an approximate numerical value).

```
(%i1) quad_qags (x^(1/2)*log(1/x), x, 0, 1, ’epsrel=1d-10);
(%o1) [0.44444444444444,4.9343245538895848E-16,315,0]
(%i2) integrate(x^(1/2)*log(1/x), x, 0, 1);
(%o2) 4/9
(%i3) float(%);
(%o3) 0.44444444444444
```

The first element of the list returned by **`quad_qags`** is the value of the numerical integral (printed with the number of digits related to the current setting of **`fpprintprec`**), the second element being an estimate of the size of the absolute error of the result, the third element **`315`** being the number of function evaluations, and the last element **`0`** being the integer error code, with **`0`** indicating no problems.

Here is an example of using Maxima’s **`quad_qagi`** for an **unbounded interval** quadrature.

```
(%i4) quad_qagi (x^2*exp(-4*x), x, 0, inf, ’epsrel=1d-8);
(%o4) [0.03125,2.9591610253764947E-11,105,0]
(%i5) integrate (x^2*exp(-4*x), x, 0, inf);
(%o5) 1/32
(%i6) float(%);
(%o6) 0.03125
```

An example of a **double integral** $\int_1^3 \left(\int_0^2 \frac{x\,y}{x+y}\, dx \right) dy$ done first with Maxima’s **`integrate`**, then with **`quad_qags`**:

```
(%i7) integrate(integrate(x*y/(x+y),x,0,2),y,1,3);
Is  y+2  positive, negative, or zero?
p;
Is  y  positive, negative, or zero?
p;
(%o7) -(35*log(5)-27*log(3)-30)/3+3*log(3)-2
```

```
(%i8) float(%);
(%o8) 2.406571818952815
(%i9) quad_qags(quad_qags(x*y/(x+y),x,0,2)[1],y,1,3);
(%o9) [2.406571818952812,2.671831443815456E-14,21,0]
(%i10) %[1];
(%o10) 2.406571818952812
```

An example of a **double integral** with a variable inner upper limit, $\int_1^3 \left(\int_0^{y/2} \frac{x\,y}{x+y}\,dx\right) dy$, done first with Maxima's **integrate**, then with **quad_qags**:

```
(%i11) integrate(integrate(x*y/(x+y),x,0,y/2),y,1,3);
Is  y  positive, negative, or zero?
p;
(%o11) -9*log(9/2)+9*log(3)+(2*log(3/2)-1)/6+9/2
(%i12) float(%);
(%o12) 0.81930239639591
(%i13) quad_qags(quad_qags(x*y/(x+y),x,0,y/2)[1],y,1,3);
(%o13) [0.81930239639591,9.0960838460930482E-15,21,0]
(%i14) %[1];
(%o14) 0.81930239639591
```

An example of using **quad_qags** for the integration of a complex function. In principle, any complex function **f** can be written as **f = fr + %i*fi**. We use the Maxima functions **realpart** and **imagpart** and the integration is done "by hand" by separately integrating the real and imaginary parts, and then combining them at the end.

```
(%i15) realpart(exp(%i*x));
(%o15) cos(x)
(%i16) imagpart(exp(%i*x));
(%o16) sin(x)
(%i17) i_real : quad_qags(realpart(exp(%i*x)),x,0,1)[1];
(%o17) 0.8414709848079
(%i18) i_imag : quad_qags(imagpart(exp(%i*x)),x,0,1)[1];
(%o18) 0.45969769413186
(%i19) i_real + %i*i_imag;
(%o19) 0.45969769413186*%i+0.8414709848079
```

1.5.5 Trapezoidal Rule for a Uniform Grid in R

See **http://en.wikipedia.org/wiki/Trapezoidal_rule** for a discussion of forms of the trapezoidal rule.

For integration over the finite interval **[a, b]**, with **N** subintervals each of size **h** and thus **h = (b - a)/N**, the trapezoidal rule is

$$\int_a^b f(x)\,dx = \frac{h}{2}\left(f(a) + 2\,f(a+h) + 2\,f(a+2\,h) + \cdots + 2\,f(b-h) + f(b)\right) \tag{1.1}$$

We will provide **R** code for the simple trapezoidal rule value of a one dimensional integral of a function or property sampled at equal intervals ("uniform grid").

Note that the **R** function **sin** turn vectors into vectors and single numbers into single numbers, and so does **x^2**, etc.

```
> yv = seq(0,1, by = 0.25); yv
[1] 0.00 0.25 0.50 0.75 1.00
> yv^2
[1] 0.0000 0.0625 0.2500 0.5625 1.0000
> f1 = function(xv) xv^2
> f1(yv)
[1] 0.0000 0.0625 0.2500 0.5625 1.0000
> sin(yv)
[1] 0.0000000 0.2474040 0.4794255 0.6816388
[5] 0.8414710
> f1(3)
[1] 9
> sin(3)
[1] 0.14112
```

With that background, here is a trapezoidal rule function for a uniform grid. In this code, **xv** is a **R**-vector of equally spaced positions where the function is to be sampled, and **yv** is a **R**-vector of function values at those positions.

```
trap = function(xv,yv){
   n = length(xv)
   (xv[2]-xv[1])*((yv[1]+yv[n])/2 + sum(yv[2:(n-1)]))}
```

Simple examples of use, first integrating $\sin(\theta)$ over the interval $[0,1]$. using only 5 data points (with θ an angle expressed in radians). We first define the **R** vector **xv**, then paste the definition of **trap** into the **RStudio** Console window, and then try out the example interactively:

```
> xv = seq(0,1,0.25); xv
[1] 0.00 0.25 0.50 0.75 1.00

> trap = function(xv,yv){
+     n = length(xv)
+     (xv[2]-xv[1])*((yv[1]+yv[n])/2 + sum(yv[2:(n-1)]))}

> trap(xv, sin(xv))
[1] 0.4573009
## using R's built-in quadrature function integrate:
> integrate(sin,0,1)
0.4596977 with absolute error < 5.1e-15

> .Last.value$value
[1] 0.4596977
```

and then integrating the same function from **1** to **0** instead:

```
> xv = seq(from = 1, to = 0, by = -0.25); xv
[1] 1.00 0.75 0.50 0.25 0.00

> trap(xv, sin(xv))
[1] -0.4573009

> integrate(sin,1,0)$value
[1] -0.4596977
```

An alternative design lets the code do the work of constructing the x-coordinate and y-coordinate lists. Let's call this **R** version **trap2**. The arguments are the name of the function, the integration interval start and end, and the desired number of subintervals **N**. The calling syntax will be **trap2(fun, a, b, N)**.

```
trap2 = function(func, a, b, N) { # N is number of panels, N+1 is length of xv
   h = (b - a)/N
   xv = seq(a, b, by = h)
   yv = func(xv)
   h*((yv[1]+yv[N+1])/2 + sum(yv[2:N]))}
```

with the behavior

```
> trap2(sin,0,1,4)
[1] 0.4573009
> trap2(sin,1,0,4)
[1] -0.4573009
```

1.5.6 Trapezoidal Rule for a Uniform Grid in Maxima

The Maxima analog of an **R** vector is a Maxima "list", denoted by square brackets, as in

```
  xv : [1,2,3,4,5,6]
```

Both Maxima and R use `xv[3]` to pick out the third element of a vector (third element of a Maxima list). The **Maxima** functions `rest` and `apply` can be used to effect the sum of part of a Maxima list of numbers.

```
(%i1) apply("+",[1,2,3]);
(%o1) 6
(%i2) rest([1,2,3]);
(%o2) [2,3]
(%i3) rest([1,2,3]-1);
(%o3) [1,2]
(%i4) xv : [1,2,3,4,5,6];
(%o4) [1,2,3,4,5,6]
(%i5) apply("+", rest(rest(xv,-1)));
(%o5) 14
(%i6) rest(rest(xv,-1));
(%o6) [2,3,4,5]
(%i7) xv[3];
(%o7) 3
```

We can use the `maxima` functions `makelist` and `map` to construct a x-coordinate list and then the corresponding y-coordinate list. We can use the `maxima` parameter `fpprintprec` to control the number of digits printed to the screen (this does not affect the 16 digit precision of the internal arithmetic).

```
(%i8) map(sin,[1,2]);
(%o8) [sin(1),sin(2)]
(%i9) float(%);
(%o9) [0.8414709848079,0.90929742682568]
(%i10) makelist(0.25*i, i, 0, 4);
(%o10) [0,0.25,0.5,0.75,1.0]
(%i11) xv : %;
(%o11) [0,0.25,0.5,0.75,1.0]
(%i12) map(sin, xv);
(%o12) [0,0.24740395925452,0.4794255386042,0.68163876002333,0.8414709848079]
(%i13) fpprintprec : 8$
(%i14) map(sin, xv);
(%o14) [0,0.247404,0.479426,0.681639,0.841471]
```

We can use the `maxima` function `quad_qags` to do a numerical check on `trap`.

```
(%i15) quad_qags(sin(x),x,0,1);
(%o15) [0.459698,5.10366964E-15,21,0]
```

with the first element of the return list being the value of the numerical integral (printed with the number of digits related to the current setting of `fpprintprec`), the second element being an estimate of the size of the absolute error of the result, the third element (21) being the number of function evaluations, and the last element (0) being the integer error code, with `0` indicating no problems.

You can pick out just the first element of this returned Maxima list using

```
(%i16) quad_qags(sin(x),x,0,1)[1];
(%o16) 0.459698
(%i17) %;
(%o17) 0.459698
```

Note the percent sign **%** recovers the previous line's output, whether a number or a list or a symbol.

An "exact value" of the integral is found in Maxima using **integrate**

```
(%i18) integrate(sin(x),x,0,1);
(%o18) 1-cos(1)
(%i19) float(%);
(%o19) 0.45969769413186
```

which has 14 accurate digits. A 19 digit "exact value" of the integral is

```
(%i20) block([fpprec:20], bfloat(1 - cos(1)));
(%o20) 4.5969769413186028260b-1
```

Here is **maxima** code for the uniform grid trapezoidal rule (we have wrapped the result with **float** to convert to a floating point number):

```
trap(xv,yv) :=
block([n :length(xv)],
  float( (xv[2] - xv[1])*((yv[1] + yv[n])/2 +
                  apply("+",rest(rest(yv,-1))))))$
```

Once we have constructed the Maxima list to be used for the first (formal) argument **xv**, we can either use **map** in the function call or else pre-define a corresponding coordinate list to be used for the second (formal) argument of **trap**.

We first paste into Maxima (the author uses the **Xmaxima** interface almost exclusively) the definition of **trap**. (Note the crucial delayed assignment symbol **:=** used to define a Maxima function.) There is one local variable declared and defined inside the local variables bracket **[ ]**, whose value remains unknown in the global environment. Once you have pasted the definition into XMaxima (using **Ctrl+v**), press the keyboard down-arrow key to get to the bottom of the input, and then press **Enter**.

```
(%i1) trap(xv,yv) :=
block([n :length(xv)],
  float( (xv[2] - xv[1])*((yv[1] + yv[n])/2 +
                  apply("+",rest(rest(yv,-1))))))$
(%i2) fpprintprec:8$
(%i3) xL : makelist(i/4,i,0,4);
(%o3) [0,1/4,1/2,3/4,1]
(%i4) yL : sin(xL);
(%o4) [0,sin(1/4),sin(1/2),sin(3/4),sin(1)]
(%i5) trap(xL, yL);
(%o5) 0.457301
```

An alternative Maxima version of the trapezoidal rule, with the syntax **trap2(fun, a, b, n)**, with n the requested number of subintervals (panels), is

```
trap2(func,a,b,n) :=
block([h : (b - a)/n,xL,yL],
    xL : makelist(a + i*h,i,0,n),
    yL : map('func, xL),
    float(h*((yL[1] + yL[n+1])/2 +
              apply("+",rest(rest(yL,-1))))))$
```

The Maxima function **float** converts numbers to floating point numbers.

```
(%i6) trap2(func,a,b,n) :=
block([h : (b - a)/n,xL,yL],
    xL : makelist(a + i*h,i,0,n),
    yL : map('func, xL),
    float(h*((yL[1] + yL[n+1])/2 +
               apply("+",rest(rest(yL,-1))))))$
(%i7) trap2(sin, 0, 1, 4);
(%o7) 0.457301
(%i8) trap2(lambda([x], sin(x)),0,1,4);
(%o8) 0.457301
(%i9) trap2(lambda([x], x*sin(x)^2),0,1,4);
(%o9) 0.208184
(%i10) quad_qags(x*sin(x)^2,x,0,1);
(%o10) [0.199694,2.21704874E-15,21,0]
```

The first example of using **trap2** uses the name of a function which Maxima knows about (either a core Maxima function, or one defined by a loaded package, or one defined by you with a name prior to calling **trap2**). The second and third examples (using the anonymous lambda function) are examples of how you could use your own (unnamed) function.

Integrating **sin(x)** instead from **x = 1** to **x = 0** should reverse the sign, and this is easy to check with version **trap2**.

```
(%i11) quad_qags(sin(x),x,1,0);
(%o11) [-0.459698,5.10366964E-15,21,0]
(%i12) trap2(sin,1,0,4);
(%o12) -0.457301
```

We have to do more work to check **trap**:

```
(%i13) xL : reverse(makelist(i/4,i,0,4));
(%o13) [1,3/4,1/2,1/4,0]
(%i14) yL : map(sin, xL);
(%o14) [sin(1),sin(3/4),sin(1/2),sin(1/4),0]
(%i15) trap(xL, yL);
(%o15) -0.457301
```

1.5.7 Trapezoidal Rule for a Non-Uniform Grid in R

Jarek Tuszynski, in the **R** package **caTools**, has a version of the trapezoidal rule which can handle unequally spaced data. (We have replaced his **as.double** with **as.numeric**, since all real floats are treated as double in **R**, and have omitted his final wrap **return()** function which is not needed; the last result in a function is what is returned (provided the interior of the function code does not contain a **return()** call which is activated by a decision).

Jarek's code uses the matrix multiply symbol **%*%** which is used to multiply matrices together, multiply a vector times a matrix, or to multiply two vectors together to form an "inner product" of two R-vectors. **R** does not distinuish row vectors from column vectors; all vectors are equal. The functions **nrow** and **ncol** return **NULL** for a vector, while the functions **NROW** and **NCOL** treat a **R** vector as a one-column matrix.

```
> yv = seq(1,3,by = 0.5); yv
[1] 1.0 1.5 2.0 2.5 3.0
> nrow(yv)
NULL
> ncol(yv)
NULL
> NCOL(yv)
[1] 1
> NROW(yv)
[1] 5
```

The inner product of two vectors is then

```
> xv = c(1,2,3); yv = c(2,2,2)
> xv %*% yv
     [,1]
[1,]   12
> as.numeric(xv %*% yv)
[1] 12
```

Without the coercive `as.numeric`, the inner product results in a one element matrix.

With that background, the non-uniform grid code is then:

```
trapz = function(xv,yv) {
   idx = 2:length(xv)
   as.numeric( (xv[idx] - xv[idx - 1]) %*% (yv[idx - 1] + yv[idx])/2)}
```

For a uniform grid, this reduces to the same algorithm used in `trap`. We can use small random numbers to deform a uniform grid into a non-uniform grid, using the `jitter` function (which adds a small amount of noise to a vector).

```
> xv = seq(0,1, by = 0.25); xv
[1] 0.00 0.25 0.50 0.75 1.00
> xv = jitter(xv)
> xv
[1] -0.03366742  0.23070275  0.50177947  0.70747696  1.01899303
> trapz = function(xv,yv) {
+     idx = 2:length(xv)
+     as.numeric( (xv[idx] - xv[idx - 1]) %*% (yv[idx - 1] + yv[idx])/2)}
> trapz(xv, sin(xv))
[1] 0.4721434
```

Because each call to the random number generator results in a different result, your results for `jitter(xv)` will not necessarily be the same as in the above.

1.5.8 Trapezoidal Rule for a Non-Uniform Grid in Maxima

The inner product of two maxima lists is produced by separating the lists by `.` .

```
(%i1) [2, 2, 2] . [1, 2, 3];
(%o1) 12
```

Again we use Maxima's `makelist` function to produce Maxima code for the non-uniform grid version of the trapezoidal rule.

```
trapz(xv,yv) :=
block([n:length(xv)],
   dxx : makelist( xv[i] - xv[i-1], i, 2, n),
   yy : makelist( yv[i-1] + yv[i], i, 2, n),
   float(dxx . yy/2))$
```

After pasting this code into Maxima, we first test this code using a uniform grid as a check.

```
(%i2) xL : makelist(i/4,i,0,4);
(%o2) [0,1/4,1/2,3/4,1]
(%i3) yL : sin(xL);
(%o3) [0,sin(1/4),sin(1/2),sin(3/4),sin(1)]
(%i4) fpprintprec:8$
(%i5) trapz(xL, yL);
(%o5) 0.457301
(%i6) trapz(xL, sin(xL));
(%o6) 0.457301
```

We now deform the uniform grid using a homemade **`jitter`** function in Maxima:

```
jitter(xv,[o]) :=
block([ampl,fac:1],
   if not listp(xv) then return("xv must be a list"),
   if length(o) > 0 then fac : o[1],
   if length(o) > 1 then ampl : o[2]
     else ampl : fac*(apply(max,xv) - apply(min,xv))/50,
   xv + ampl*makelist(random(2.0) -1, i, 1, length(xv)))$
```

The syntax is **`jitter(alist [factor, amplitude])`** in which factor and amplitude are both optional arguments with default values. The default value of factor is **`1`**. To override the default value of factor, use **`jitter(alist,myfactor)`**. To override the default value of amplitude, you must have a value in the factor slot also. Here are some simple examples:

```
(%i7) jitter([1,2,3]);
(%o7) [0.997476,2.0216678,3.0163642]
(%i8) jitter([1,2,3],2);
(%o8) [1.0025236,2.0397766,2.9712082]
(%i9) jitter([1,2,3],1,1/50);
(%o9) [0.998108,1.9851285,3.0069044]
(%i10) jitter(2);
(%o10) "xv must be a list"
```

We now use **`jitter`** to deform the previously used uniform grid and again use **`trapz`**:

```
(%i11) xL;
(%o11) [0,1/4,1/2,3/4,1]
(%i12) xL : jitter(xL);
(%o12) [-0.0193452,0.261299,0.490535,0.736513,1.0169432]
(%i13) yL : map (sin, xL);
(%o13) [-0.019344,0.258336,0.471098,0.671709,0.850504]
(%i14) trapz(xL, yL);
(%o14) 0.471132
```

1.5.9 Simpson's 1/3 Rule in R

See **`http://en.wikipedia.org/wiki/Simpson%27s_rule`** for a discussion of Simpson's rule.

For integration over the finite interval **`[a, b]`**, with **`N`** the **even** number of subintervals, each of size **`h`** and thus **`h = (b - a)/N`**, Simpson's $1/3$ rule is

$$\int_a^b f(x)\,dx = \frac{h}{3}\left(f(a) + 4\,f(a+h) + 2\,f(a+2\,h) + 4\,f(a+3\,h) + \cdots + 4\,f(b-h) + f(b)\right) \qquad (1.2)$$

Note that we assume a "uniform grid", and the interval of integration is divided into an even number of subintervals, so the length of the position vector **`xv`** should be odd. A "vectorized" **`R`** code with similarities to **`trap`** above is (with **`N`** the even number of subintervals)

```
simp = function(xv,yv) {
   N = length(xv) - 1
   if (N %% 2 != 0) {
        return(" length(xv) should be an odd integer ")}
   h = xv[2] - xv[1]
   if (N == 2) s = yv[1]+4*yv[2]+yv[3]  else {
       s = yv[1] + yv[N+1] + 4*sum(yv[seq(2,N,by=2)]) +
                2*sum(yv[seq(3,N-1,by=2)])}
   s*h/3}
```

After pasting this code into the **R** Console, we get

```
> xv = seq(0,1,by = 0.25); xv
[1] 0.00 0.25 0.50 0.75 1.00
> simp(xv, sin(xv))
[1] 0.4597077
> integrate(sin,0,1)
0.4596977 with absolute error < 5.1e-15
```

We also check the case of integration from **1** to **0**:

```
> xv = seq(1,0,by = -0.25); xv
[1] 1.00 0.75 0.50 0.25 0.00
> simp(xv, sin(xv))
[1] -0.4597077
> integrate(sin,1,0)
-0.4596977 with absolute error < 5.1e-15
```

and check the case **N = 2**

```
> xv = c(0, 0.5, 1); xv
[1] 0.0 0.5 1.0
> simp(xv, sin(xv))
[1] 0.4598622
```

and finally check the error return if the length of **xv** is not odd:

```
> xv = seq(0,1.25, by = 0.25); xv
[1] 0.00 0.25 0.50 0.75 1.00 1.25
> simp(xv, sin(xv))
[1] " length(xv) should be an odd integer "
```

A second **R** language version of Simpson's rule with the syntax **simp2(fun, x1, x2, num_panel)** requires the creation of the "position vector" **xv** inside the function. We then have two ways of using **seq**. One method uses the **by** argument, and the other uses the **length.out** argument (which can be shortened to **length**). Here is interactive experimentation in **R**:

```
> a = 0.5; b = 1.5
> N = 4; h = (b-a)/N; h
[1] 0.25
> seq(a,b,by=h)
[1] 0.50 0.75 1.00 1.25 1.50
> seq(a,b,length = 5)
[1] 0.50 0.75 1.00 1.25 1.50
```

Here is **R** code for **simp2**:

```
simp2 = function(func, a, b, N) {
    if (N %% 2 != 0) {
        return(" N should be an even integer ")}
    h = (b - a)/N
    xv = seq(a, b, by=h)
    yv = func(xv)
    if (N == 2) s = yv[1]+4*yv[2]+yv[3]  else {
          s = yv[1] + yv[N+1] + 4*sum(yv[seq(2,N,by=2)]) +
                2*sum(yv[seq(3,N-1,by=2)])}
    s*h/3}
```

with the behavior:

```
> simp2(sin,0,1,4)
[1] 0.4597077
> simp2(sin,1,0,4)
[1] -0.4597077
> simp2(sin,0,1,5)
[1] " N should be an even integer "
```

1.5.10 Simpson's 1/3 Rule in Maxima

A straightforward translation of the **`R`** version of **`simp`** into the Maxima language is:

```
simp(xv, yv) :=
block([n, s],
   n : length(xv) - 1,  /* number of panels */
   if mod(n,2) # 0 then return(" length(xv) should be an odd integer "),
   h : xv[2] - xv[1],
   if n = 2 then s : yv[1] + 4*yv[2] + yv[3]
   else   s : yv[1] + yv[n+1] +
           4*apply("+",makelist(yv[i],i,2,n,2)) +
           2*apply("+", makelist(yv[i],i,3,n-1,2)),
   float(s*h/3))$
```

Integrating **`sin`** first over **`[0,1]`** as before, with 4 panels, and then from **`1`** back to **`0`**:

```
(%i1) xL : makelist(i/4,i,0,4);
(%o1) [0,1/4,1/2,3/4,1]
(%i2) yL : map(sin, xL);
(%o2) [0,sin(1/4),sin(1/2),sin(3/4),sin(1)]
(%i3) fpprintprec:8$
(%i4) simp(xL,yL);
(%o4) 0.459708
(%i5) xL : reverse(xL);
(%o5) [1,3/4,1/2,1/4,0]
(%i6) yL : map('sin, xL);
(%o6) [sin(1),sin(3/4),sin(1/2),sin(1/4),0]
(%i7) simp(xL,yL);
(%o7) -0.459708
```

and finally a test of the error return feature:

```
(%i8) xL : makelist(i/4,i,0,5);
(%o8) [0,1/4,1/2,3/4,1,5/4]
(%i9) length(xL);
(%o9) 6
(%i10) yL : map('sin, xL);
(%o10) [0,sin(1/4),sin(1/2),sin(3/4),sin(1),sin(5/4)]
(%i11) simp(xL,yL);
(%o11) " length(xv) should be an odd integer "
```

And here is a second Maxima version of Simpson's rule with the syntax **`simp2(fun, x1, x2, num_panel)`**.

```
simp2(func,a,b,n) :=
block([h,xL, yL, s],
    if mod(n,2) # 0 then return(" n should be even number of panels"),
    h : (b - a)/n,
    xL : makelist(a + i*h,i,0,n),
    yL : map('func, xL),
    if n = 2 then s : yL[1] + 4*yL[2] + yL[3]
    else  s : yL[1] + yL[n+1] +
           4*apply("+",makelist(yL[i],i,2,n,2)) +
           2*apply("+", makelist(yL[i],i,3,n-1,2)),
    float(s*h/3))$
```

with the behavior:

```
(%i12) simp2(sin,0,1,4);
(%o12) 0.459708
(%i13) simp2(sin,1,0,4);
(%o13) -0.459708
(%i14) simp2(sin,0,1,5);
(%o14) " n should be even number of panels"
```

1.5.11 Checking Integrals with the Wolfram Alpha Webpage

The Wolfram Alpha web page, **`http://www.wolframalpha.com`**, allows free one-line commands (integrals, derivatives, plots, etc.,) which may require translating Maxima syntax into Mathematica syntax.

A web page with some translation from Maxima to Mathematica is

```
     http://www.math.harvard.edu/computing/maxima/
```

A larger comparison of Maxima, Maple, and Mathematica syntax is at

```
     http://beige.ucs.indiana.edu/P573/node35.html
```

Maxima syntax for 1d symbolic integration over a specified interval is

```
(%i13) integrate(cos(x),x,0,1);
(%o13) sin(1)
(%i14) float(%);
(%o14) 0.8414709848079
```

The corresponding Mathematica request would be (note the curly brackets):

```
     Integrate[Cos[x], {x, 0, 1}]
```

for a symbolic answer, and

```
     NIntegrate[Cos[x], {x, 0, 1}]
```

produces a numerical result corresponding to Maxima's

```
     float(integrate(cos(x),x,0,1))
```

An integration over an unbounded interval, done in Maxima with (for example)

```
     float(integrate(xˆ2*exp(-x),x,0,inf))
```

is done using Mathematica with

```
     NIntegrate[xˆ2*Exp[-x], {x, 0, Infinity}]
```

Mathematica also has the function **N** which computes the numerical value of **expr** with the syntax:

```
    N[ expr ]
```

or

```
    expr //N
```

Two examples of numerical two dimensional integration in Mathematica syntax are:

```
    NIntegrate[1/Sqrt[x + y], {x,0,1},{y,0,1} ]
    NIntegrate[Sin[ x*y ],{x,0,1}, {y,0,1} ]
```

1.5.12 Dealing with an Integral with Infinite or Very Large Limits

To compute an integral over a semi-infinite interval $[\mathbf{a}, \infty]$, change variables: $\mathbf{x} = \mathbf{a} + \mathbf{t}/(\mathbf{1} - \mathbf{t})$.

$$\int_{a}^{\infty} f(x)\,dx = \int_{0}^{1} f\left(a + \frac{t}{1-t}\right)\frac{1}{(1-t)^2}\,dt \tag{1.3}$$

Example 1: $\int_1^{\infty} x^2\, e^{-x}\, dx = \int_0^1 \frac{1}{(1-t)^4}\,\exp\left(\frac{-1}{1-t}\right)\,dt$

We compute the left hand side, first with Maxima

```
(%i3) integrate(x^2*exp(-x),x,1,inf);
(%o3) gamma_incomplete(3,1)
(%i4) float(%);
(%o4) 1.839397205857212
```

and then with **R**

```
> integrate(function(x) x^2*exp(-x),1, Inf)
1.839397 with absolute error < 3.1e-05
```

We then check the right hand side, first with Maxima

```
(%i6) quad_qags(exp(-(1/(1-t)))/(1-t)^4,t,0,1);
(%o6) [1.839397205857212,1.0649276645161734E-12,189,0]
```

and then with **R**

```
> integrate(function(t) exp(-(1/(1-t)))/(1-t)^4,0,1)
1.839397 with absolute error < 6.3e-05
```

Example 2: $\int_{-1}^{\infty} x^2\, e^{-x}\, dx = \int_0^1 \frac{(2\,t-1)^2}{(1-t)^4}\,\exp\left(\frac{-(2\,t-1)}{1-t}\right)\,dt$

The left hand side, via Maxima

```
(%i7) integrate(x^2*exp(-x),x,-1,inf);
(%o7) %e
(%i8) float(%);
(%o8) 2.718281828459045
```

and with **R**

```
> integrate(function(x) x^2*exp(-x),-1,Inf)
2.718282 with absolute error < 0.00016
```

and the right hand side with Maxima

```
(%i9) quad_qags((2*t - 1)^2*exp(-(2*t-1)/(1-t))/(1-t)^4,t,0,1);
(%o9) [2.718281828459046,1.4810265488268963E-11,189,0]
```

and then with **R**

```
> integrate(function(t) (2*t - 1)^2*exp(-(2*t-1)/(1-t))/(1-t)^4,0,1)
2.718282 with absolute error < 4.5e-07
```

To compute an integral over the unbounded interval $[-\infty, \infty]$, change variables: $x = t/(1 - t^2)$.

$$\int_{-\infty}^{\infty} f(x)\,dx = \int_{-1}^{1} f\left(\frac{t}{1-t^2}\right)\frac{1+t^2}{(1-t^2)^2}\,dt \tag{1.4}$$

Example: $\int_{-\infty}^{\infty} x^2\, e^{-|x|}\,dx = \int_{-1}^{1} \frac{t^2\,(1+t^2)}{(1-t^2)^4}\,\exp\left(-\left|\frac{t}{1-t^2}\right|\right)\,dt$

We check the left-hand side, first with Maxima

```
(%i10) integrate(x^2*exp(-abs(x)),x,-inf,inf);
(%o10) 4
```

and then with **R**

```
> integrate(function(x) x^2*exp(-abs(x)),-Inf,Inf)
4 with absolute error < 0.00014
```

We next check the right-hand side, first with Maxima

```
(%i11) quad_qags(t^2*(1+t^2)*exp(-abs(t/(1-t^2)))/(1-t^2)^4, t, -1, 1);
(%o11) [4.0,2.9692915364436538E-12,483,0]
```

and then with **R**

```
> integrate(function(t) t^2*(1+t^2)*exp(-abs(t/(1-t^2)))/(1-t^2)^4, -1, 1)
4 with absolute error < 0.00016
```

There is a discussion of these two transformation formulas, on the webpage **`http://ab-initio.mit.edu/wiki/index.php/Cubature`** in the context of the **C** implementation of the cubature code for **`adaptIntegral`**.

> Note the Jacobian factors multiplying $f(\cdots)$ in both integrals, and also that the limits of the t integrals are different in the two cases.
>
> In multiple dimensions, one simply performs this change of variables on each dimension separately, as desired, multiplying the integrand by the corresponding Jacobian factor for each dimension being transformed.
>
> The Jacobian factors diverge as the endpoints are approached. However, if $f(x)$ goes to zero at least as fast as $1/x^2$, then the limit of the integrand (including the Jacobian factor) is finite at the endpoints. If your $f(x)$ vanishes more slowly than $1/x^2$ but still faster than $1/x$, then the integrand blows up at the endpoints but the integral is still finite (it is an integrable singularity), so the code will work (although it may take many function evaluations to converge). If your $f(x)$ vanishes only as $1/x$, then it is not absolutely convergent and much more care is required even to define what you are trying to compute. (In any case, the h-adaptive quadrature/cubature rules currently employed in cubature.c do not evaluate the integrand at the endpoints, so you need not implement special handling for $|t| = 1$)

To compute an integral over a semi-infinite interval $[-\infty, b]$, change variables: $x = b + t/(1+t)$ to get

$$\int_{-\infty}^{b} f(x)\, dx = \int_{-1}^{0} f\left(b + \frac{t}{1+t}\right) \frac{1}{(1+t)^2}\, dt \tag{1.5}$$

Example 1: $\int_{-\infty}^{-1} x^2\, e^x\, dx = \int_{-1}^{0} \frac{1}{(1+t)^4} \exp\left(\frac{-1}{1+t}\right)\, dt$

We compute the left hand side, first with Maxima

```
(%i14) integrate(x^2*exp(x),x,-inf,-1);
(%o14) gamma_incomplete(3,1)
(%i15) float(%);
(%o15) 1.839397205857212
```

and next with **R**

```
> integrate(function(x) x^2*exp(x),-Inf,-1)
1.839397 with absolute error < 3.1e-05
```

We then compute the right-hand side, first with Maxima

```
(%i16) quad_qags(exp(-1/(1+t))/(1+t)^4, t, -1, 0);
(%o16) [1.839397205857212,1.0649276645161734E-12,189,0]
```

and then with **R**

```
> integrate(function(t) exp(-1/(1+t))/(1+t)^4,  -1, 0)
1.839397 with absolute error < 6.3e-05
```

Example 2: $\int_{-\infty}^{1} x^2\, e^x\, dx = \int_{-1}^{0} \frac{(1+2\,t)^2}{(1+t)^4} \exp\left(\frac{1+2\,t}{1+t}\right)\, dt$

We check the left-hand side with Maxima

```
(%i17) integrate(x^2*exp(x),x,-inf,1);
(%o17) %e
(%i18) float(%);
(%o18) 2.718281828459045
```

and with **R**

```
> integrate(function(x) x^2*exp(x),-Inf,1)
2.718282 with absolute error < 0.00016
```

We check the right-hand side with Maxima

```
(%i19) quad_qags((1+2*t)^2*exp((1+2*t)/(1+t))/(1+t)^4,t,-1,0);
(%o19) [2.718281828459046,1.4810265488268963E-11,189,0]
```

and then with **R**

```
> integrate(function(t)  (1+2*t)^2*exp((1+2*t)/(1+t))/(1+t)^4,-1,0)
2.718282 with absolute error < 4.5e-07
```

A more general transformation formula, which can be used with either unbounded integral limits or simply very large limits (on the scale length of the integrand) is the identity (provided the product $a\, b\ > 0$):

$$\int_{a}^{b} f(x)\, dx = \int_{1/b}^{1/a} \frac{1}{t^2}\, f\left(\frac{1}{t}\right)\, dt \tag{1.6}$$

This identity can be used with *either* $b \to \infty$ with a positive, *or* with $a \to -\infty$ and b negative provided $f(x)$ decreases toward infinity faster than $1/x^2$.

For example, one can split the integral $\int_0^\infty \mathbf{f}(\mathbf{x})\,\mathbf{dx} = \int_0^\mathbf{c} \mathbf{f}(\mathbf{x})\,\mathbf{dx} + \int_\mathbf{c}^\infty \mathbf{f}(\mathbf{x})\,\mathbf{dx}$, choosing the breakpoint **c** to be large enough that the asymptotic limit of the integrand is being approached, and then transform the *second* term into $\int_0^{1/\mathbf{c}} \frac{1}{\mathbf{t}^2}\,\mathbf{f}\left(\frac{1}{\mathbf{t}}\right)\,\mathbf{dt}$.

Example taking $\mathbf{c} = 20$:

$$\int_0^\infty \cos(\mathbf{x})\,\mathbf{e}^{-\mathbf{x}}\,\mathbf{dx} = \int_0^{20} \cos(\mathbf{x})\,\mathbf{e}^{-\mathbf{x}}\,\mathbf{dx} + \int_{20}^\infty \cos(\mathbf{x})\,\mathbf{e}^{-\mathbf{x}}\,\mathbf{dx}$$

and we use the above transformation on the second term to get an integral over a finite domain:
$\int_{20}^\infty \cos(\mathbf{x})\,\mathbf{e}^{-\mathbf{x}}\,\mathbf{dx} = \int_0^{1/20} \frac{\cos(1/\mathbf{t})\,\exp(-1/\mathbf{t})}{\mathbf{t}^2}\,\mathbf{dt}$.

Using Maxima for the original integral with unbounded limits

```
(%i1) integrate(cos(x)*exp(-x),x,0,inf);
(%o1) 1/2
```

we then use **R** to show that the finite domain transformed term is negligible, and only the first term including contributions from $\mathbf{x} < 20$ needs to be retained.

```
> integrate(function(t) cos(1/t)*exp(-1/t)/t^2,0,0.05)
-5.203003e-10 with absolute error < 1.5e-14
> integrate(function(x) cos(x)*exp(-x),0,20)
0.5 with absolute error < 4.6e-05
```

This split-up method can be used to evaluate an integral to the precision desired by simply choosing **c** large enough that the neglected piece is small enough compared to the dominant term over `[0, c]`.

1.5.13 Handling Integrable Singularities at Endpoints

It is easy to find the integral $\int_0^\pi \frac{\mathbf{dx}}{\sqrt{\mathbf{x}}} = \int_0^\pi \frac{\mathbf{d}}{\mathbf{dx}}\left(2\sqrt{\mathbf{x}}\right)\,\mathbf{dx} = 2\sqrt{\pi}$ despite the singularity of the integrand at $\mathbf{x} = 0$. This is an example of an “integrable singularity”. Likewise Maxima

```
(%i2) integrate(1/sqrt(x),x,0,%pi);
(%o2) 2*sqrt(%pi)
(%i3) float(%);
(%o3) 3.544907701811032
(%i4) quad_qags(1/sqrt(x),x,0,%pi);
(%o4) [3.544907701811034,8.8817841970012523E-15,231,0]
```

and **R** have no difficulties.

```
> options(digits=16)
> integrate(function(x) 1/sqrt(x),0,pi)$val
[1] 3.544907701811033
```

Multiplying the above integrand by the well behaved function $\cos(\mathbf{x})$ should not increase the numerical effort, first with **R**

```
> integrate(function(x) cos(x)/sqrt(x),0,pi)$val
[1] 1.325734626520552
```

and then with Maxima

```
(%i5) quad_qags(cos(x)/sqrt(x),x,0,%pi);
(%o5) [1.325734626520542,2.9753977059954195E-13,315,0]
```

Maxima can return an analytic result in terms of **erf** which represents the Error Function, defined in Sec. 7.1.1 of Abramowitz and Stegun: **Handbook of Mathematical Functions**.

$$\mathrm{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2}\, dt \tag{1.7}$$

But converting Maxima's analytic result to a floating point number requires discussion. We know the numerical result must be a real number.

```
(%i6) ival : integrate(cos(x)/sqrt(x),x,0,%pi);
(%o6) -sqrt(%pi)*((sqrt(2)*%i-sqrt(2))*erf(sqrt(%pi)*(sqrt(2)*%i+sqrt(2))/2)
                  +(sqrt(2)*%i+sqrt(2))*erf(sqrt(%pi)*(sqrt(2)*%i-sqrt(2))/2))/4
(%i7) float(ival);
(%o7) -0.4431134627263*((1.414213562373095*%i-1.414213562373095)
                        *erf(0.88622692545276
                              *(1.414213562373095*%i+1.414213562373095))
                        +(1.414213562373095*%i+1.414213562373095)
                         *erf(0.88622692545276
                               *(1.414213562373095*%i-1.414213562373095)))
(%i8) expand(%);
(%o8) 1.9678190753608281E-16*%i+1.325734626520541
(%i9) realpart(%);
(%o9) 1.325734626520541
```

The small imaginary part is due to the approximate numerical evaluation of the Error Function. Experience has shown that reducing a complicated expression involving **%i** (which stands for $\sqrt{-1}$) to a possibly complex number is simplified by defining the homemade function ("cfloat": complex float)

```
    cfloat(ee):= expand(float(rectform(ee)))$
```

```
(%i10) cfloat(ee):= expand(float(rectform(ee)))$
(%i11) cfloat(ival);
(%o11) 2.6089841481297814E-16*%i+1.325734626520542
(%i12) realpart(%);
(%o12) 1.325734626520542
```

The Maxima function **rectform** returns an expression of the form **a + b*%i**, in which **a** and **b** are real.

```
(%i13) rectform(exp(%i*x));
(%o13) %i*sin(x)+cos(x)
(%i14) rectform(cos(%i*x));
(%o14) cosh(x)
```

If our numerical quadrature software had been unable to deal with the integral $\int_0^b \frac{\cos(x)}{\sqrt{x}}\, dx$ we could have tried **integration by parts**, using $\frac{\cos(x)}{\sqrt{x}} = \frac{d}{dx}(2\sqrt{x}\cos(x)) - 2\sqrt{x}\frac{d}{dx}\cos(x)$. Then $\int_0^b \frac{\cos(x)}{\sqrt{x}}\, dx = 2\sqrt{b}\cos(b) + 2\int_0^b \sqrt{x}\sin(x)\, dx$, and the remaining integral is not singular.

Quoting Joel Ferziger: **Numerical Methods for Engineering Application**, p. 47,

> In many singular integrals it is possible to factor the integrand into the product of two components: one that contains the singularity but is easily integrated analytically, and a second that is not singular and can be differentiated. In such a case integration by parts will often convert the integral into one that is not singular . . . sometimes it is desirable to repeat the integration by parts to obtain a still smoother integral.

Ferziger then suggests a second strategy called **singularity subtraction**:

> A related but slightly different method is to factor the integral into the sum of two parts: one that contains the singularity but is integrable analytically, and a second that is nonsingular but requires numerical quadrature.

Here is a general example:

$$\int_0^b \frac{f(x)}{\sqrt{x}}\,dx = \int_0^b \frac{f(x) - f(0)}{\sqrt{x}}\,dx + f(0)\int_0^b \frac{dx}{\sqrt{x}} \tag{1.8}$$

Applied to our example, this becomes $\int_0^b \frac{\cos(x)}{\sqrt{x}}\,dx = \int_0^b \frac{dx}{\sqrt{x}} + \int_0^b \frac{\cos(x)-1}{\sqrt{x}}\,dx$.

Because $(1 - x^2) = (1 - x)(1 + x) \to 2(1 - x)$ as $x \to 1$, the method of singularity subtraction can be used to replace the integral $\int_0^1 \frac{dx}{\sqrt{1-x^2}}$ by the two terms $\int_0^1 \left(\frac{1}{\sqrt{1-x^2}} - \frac{1}{\sqrt{2(1-x)}}\right) dx + \int_0^1 \frac{dx}{\sqrt{2(1-x)}}$, in which the second term is easy to integrate, and the integrand of the first term is smooth as $x \to 1$.

However, both Maxima and **R** have no difficulties with this last example, first Maxima:

```
(%i15) quad_qags(1/sqrt(1-x^2),x,0,1);
(%o15) [1.570796326794881,5.4368065605103766E-11,315,0]
(%i16) integrate(1/sqrt(1-x^2),x,0,1);
(%o16) %pi/2
```

and then **R**:

```
> integrate(function(x) 1/sqrt(1-x^2),0,1)
1.570796326786759 with absolute error < 1.7e-06
```

1.6 Finding Roots

The root of a function $f(x)$ is the value of x for which $f(x) = 0$. In **Computational Physics**, Ch.1, Sec.3, Koonin discusses the elementary one variable root finding methods known as the bisection method, the Newton-Raphson method, and the secant method. See `http://en.wikipedia.org/wiki/Bisection_method`, `http://en.wikipedia.org/wiki/Secant_method`, and `http://en.wikipedia.org/wiki/Newton-Raphson`.

1.6.1 R Function uniroot

The core **R** function `uniroot`, has the syntax `uniroot(func, interval, ...)`. Documentation is at `http://127.0.0.1:11976/library/stats/html/uniroot.html`.

If `f` is the given function, `uniroot` searches for a value of `x` such that `f(x)` is a sufficiently close to zero and returns the first such `x`. `uniroot` assumes the given function is continuous in the given interval and that the given function has at least one root in the interval supplied. That interval is searched starting from the lower end of the interval.

If the given function does not have opposite signs at the ends of the given interval, `uniroot` returns an error message.

```
> uniroot(function(x) (x - 1)^2,c(0,2))
Error in uniroot(function(x) (x - 1)^2, c(0, 2)) :
  f() values at end points not of opposite sign
```

We use `uniroot` to search for the single root of the equation $x^2 - 5 = 0$ in the interval $[1, 5]$. We can make a simple plot of the function `f(x) = x^2 - 5` as follows:

```
> fun = function(x) x^2 - 5
> curve(fun,-5,5,lwd = 2)
> abline(h = 0, lty = 3)
> abline(v = 0, lty = 3)
```

The `lwd` parameter setting doubles the line width. The `abline` statements add a horizontal line at `y = 0` to the figure created by `curve` and then a vertical line at `x = 0` to add "axes". The `lty` setting is a line type setting.

`uniroot` returns a list with four elements: the approximate root location `$root`, the value of the given function at the root location found (should be close to zero) `f.root`, the number of iterations required `$iter`, and the precision estimate `$estim.prec`.

```
> ?uniroot
> exact = sqrt(5); exact
[1] 2.23606797749979
> uniroot(fun, c(1, 5))
$root
[1] 2.236067654886587
$f.root
[1] -1.442770001247595e-06
$iter
[1] 7
$estim.prec
[1] 6.103515625088818e-05
> numr1 = uniroot(fun, c(1, 5))$root; numr1
[1] 2.236067654886587
> (numr1 - exact)/exact
[1] -1.442770105870001e-07
```

The default value of `tol` is roughly `1e-4`; actually it is given by `.Machine$double.eps^0.25`:

```
> .Machine$double.eps^0.25
[1] 0.0001220703125
> numr2 = uniroot(fun, c(1, 5),tol=1e-8)$root; numr2
[1] 2.236067977499836
> (numr2 - exact)/exact
[1] 2.065468415501731e-14
```

1.6.2 R function polyroot

The base code of **R** also includes `polyroot`, designed to find zeros of a real or complex polynomial.

1.6.3 R function rootSolve::uniroot.all

The **rootSolve** package function `uniroot.all` is a simple extension of `uniroot` which extracts many (presumably *all*) roots in the interval. Here is an example taken from Sec.1.1 of **rootSolve.pdf** by Karline Soetaert.

To find the root of function $\mathbf{f(x) = \cos^3(2\,x)}$ in the interval $[\mathbf{0}, \mathbf{8}]$ and plot the curve, we write:

```
> fun = function (x) cos(2*x)^3
> curve(fun, 0, 8, lwd = 2)
> abline(h = 0, lty = 3)
> uni = uniroot(fun, c(0, 8))$root; uni
[1] 3.927016369513807
> points(uni, 0, pch = 16, cex = 2)
```

Although the graph (figure 1) clearly demonstrates the existence of many roots in the interval $[\mathbf{0}, \mathbf{8}]$, the **R** function `uniroot` extracts only one. Here we load the package **rootSolve** and use `uniroot.all`, which finds five roots in the given interval. We then place markers in the plot at all these roots.

```
> library(rootSolve)
> all.roots = uniroot.all(fun, c(0, 8)); all.roots
[1] 0.7853994191100704 2.3561753389863349 3.9270077504814251 5.4977787040656763
[5] 7.0685536633583075
> points(all.roots, y = rep(0, length(all.roots)), pch = 16, cex = 2)
```

Quoting Soetaert

> ...`uniroot.all` does that by first subdividing the interval into small sections and, for all sections where the function value changes sign, invoking `uniroot` to locate the root.
>
> Note that this is not a full-proof method: in case subdivision is not fine enough some roots will be missed. Also, in case the curve does not cross the X-axis, but just "touches" it, the root will not be retrieved; (but neither will it be located by `uniroot`).

1.6.4 R Newton-Raphson: newton

The Newton-Raphson root finding algorithm is

$$v^{k+1} = v^k - \frac{f(v^k)}{f'(v^k)}, \tag{1.9}$$

To implement this iteration, the symbolic derivative in the denominator is replaced by a centered numerical derivative `s`. The initial guess `x0` needs to be "close enough" to the root for the Newton-Raphson algorithm to be successful. The criterion for returning the iterate `xn` is that the absolute value of the given function evaluated at `xn` be less than `eps`. We also want to avoid dividing by zero in a graceful way.

```
newton = function(f,x0,eps = 1e-5,h = 1e-4,small = 1e-14) {
    xn = x0
    repeat {
       if ( abs(f(xn)) < eps ) return(xn)
       s = (f(xn+h) - f(xn-h))/(2*h)
       if (abs(s) <= small) {
          cat(" derivative at",xn,"is zero \n")
          break}
       xn = xn - f(xn)/s}}
```

As an example, we search for the root of `sin(x)` near `x = 3`, which is `pi`, decreasing the value of eps (which has a default value built into the code).

```
> newton(sin,3)
[1] 3.141592653298889
> r1 = newton(sin,3,eps=1e-8); r1
[1] 3.141592653298889
> pi
[1] 3.141592653589793
> (r1 - pi)
[1] -2.909041896259623e-10
> (r1 - pi)/pi
[1] -9.259767949022792e-11
> r2 = newton(sin,3,eps=1e-12); r2
[1] 3.141592653589793
> (r2 - pi)
[1] 0
```

We next use `newton` to find the positive root of the function $x^2 - 5$, which is $\sqrt{5}$.

```
> exact = sqrt(5); exact
[1] 2.23606797749979
> nr = newton(function(x) x^2-5,1); nr
[1] 2.236068895643369
> (nr - exact)
[1] 9.181435789429315e-07
> nr = newton(function(x) x^2-5,1,eps = 1e-8); nr
[1] 2.236067977499978
```

```
> (nr - exact)
[1] 1.882938249764265e-13
> nr = newton(function(x) x^2-5,1,eps = 1e-12); nr
[1] 2.236067977499978
> (nr - exact)
[1] 1.882938249764265e-13
> nr = newton(function(x) x^2-5,1,eps = 1e-12,h=1e-8); nr
[1] 2.236067977499974
> (nr - exact)
[1] 1.84297022087776e-13
```

1.6.5 R function secant

The secant root finding algorithm is

$$v^{k+1} = v^k - f\left(v^k\right) \frac{v^k - v^{k-1}}{f\left(v^k\right) - f\left(v^{k-1}\right)} \tag{1.10}$$

The secant method requires two initial guesses **x0** and **x1** to get started.

```
secant = function(f,x0,x1,eps = 1e-5, small = 1e-14) {
    repeat {
      if ( abs(f(x1)) < eps ) return(x1)
      s = f(x1) - f(x0)
      if (abs(s) <= small) {
           cat(" derivative near",(x0+x1)/2,"is zero \n")
           break}
      x2 = x1 - f(x1)*(x1 - x0)/s
      x0 = x1
      x1 = x2}}
```

We use **secant** to search for the root of **sin(x)** near **x = 3**.

```
> secant(sin,2,3)
[1] 3.141592682798589
> sr = secant(sin,2,3,eps=1e-8); sr
[1] 3.141592653589793
> (sr - pi)
[1] 0
```

We next use **secant** to find the positive root of the function $x^2 - 5$, which is $\sqrt{5}$.

```
> exact = sqrt(5); exact
[1] 2.23606797749979
> secant(function(x) x^2-5,1,2)
[1] 2.236068111455108
> sr = secant(function(x) x^2-5,1,2,eps=1e-8);sr
[1] 2.236067977496407
> (sr - exact)
[1] -3.382627511427927e-12
> sr = secant(function(x) x^2-5,1,2,eps=1e-12);sr
[1] 2.23606797749979
> (sr - exact)
[1] 0
```

1.6.6 Maxima functions find_root and bf_find_root

For roots of general functions of one variable, Maxima has **find_root**, **bf_find_root**, and **newton**. The author has also written a bigfloat version of newton, called **bfnewton** (see later). For roots of polynomials, Maxima has **realroots**, **allroots**, and **bfallroots**.

The Maxima function **find_root** has a behavior which is similar to the **R** function **uniroot**. The documentation for the Maxima functions **find_root** and the bigfloat version **bf_find_root** is (with editing):

```
find_root (expr, var, a, b, [abserr, relerr])
find_root (f, a, b, [abserr, relerr])
bf_find_root (expr, var, a, b, [abserr, relerr])
bf_find_root (f, a, b, [abserr, relerr])
```

If the form **find_root(expr,var,...)** is used, the symbol used for **var** should be the same symbol as is used in **expr**. For example **find_root(y^2 - 5, y, 0,5)**. **find_root** 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)**. **find_root** returns an error if **f(a)** has the same sign as **f(b)**.

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.

find_root initially applies binary search. If the function in question appears to be smooth enough, **find_root** applies linear interpolation instead.

bf_find_root is a bigfloat version of **find_root**. The function is computed using bigfloat arithmetic and a bigfloat result is returned. Otherwise, **bf_find_root** is identical to **find_root**, and the following description is equally applicable to **bf_find_root**.

The accuracy of **find_root** is governed by the keywords **abserr** and **relerr**, which are optional keyword arguments to **find_root**. These keyword arguments take the form **keyword = value**. These keyword arguments are: 1. **abserr**: Desired absolute error of function value at root. The default value is the value of the global parameter **find_root_abs**.

```
(%i1) find_root_abs;
(%o1) 0.0
```

2. **relerr**: Desired relative error of root. The default value is the value of the global parameter **find_root_rel**.

```
(%i2) find_root_rel;
(%o2) 0.0
```

find_root stops when the function in question evaluates to something less than or equal to **abserr**, or if successive approximants **var_0**, **var_1** differ by no more than **relerr * max(abs(var_0), abs(var_1))**. The default values of **find_root_abs** and **find_root_rel** are both zero.

find_root expects the function in question to have a different sign at the endpoints of the search interval. When the function evaluates to a number at both endpoints and these numbers have the same sign, the behavior of **find_root** is governed by **find_root_error**. When **find_root_error** is **true**, **find_root** prints an error message. Otherwise **find_root** returns the value of **find_root_error**. The default value of **find_root_error** is **true**.

```
(%i3) find_root_error;
(%o3) true
```

If **expr** or **f** evaluates to something other than a number at any step in the search algorithm, **find_root** returns a partially-evaluated **find_root** expression.

The order of **a** and **b** is ignored; the region in which a root is sought is **[min(a, b), max(a, b)]**.

In the following examples, we accept the default **find_root** accuracy parameter values. A simple test of accuracy is to search for the positive root of the function $\mathbf{f(x) = x^2 - 5}$, located at $\mathbf{x_r = \sqrt{5}}$.

```
(%i4) exact : block([fpprec:20], bfloat(sqrt(5)));
(%o4) 2.2360679774997896964b0
(%i5) r1 : find_root(x^2 - 5,x,0,5);
(%o5) 2.23606797749979
(%i6) block([fpprec:20], bfloat(r1 - exact));
(%o6) 1.0864383394662557869b-16
```

Different syntax choices are illustrated by the first help manual examples

```
(%i7) f(x) := sin(x) - x/2;
(%o7) f(x):=sin(x)-x/2
(%i8) find_root (sin(x) - x/2, x, 0.1, %pi);
(%o8) 1.895494267033981
(%i9) find_root (sin(x) = x/2, x, 0.1, %pi);
(%o9) 1.895494267033981
(%i10) find_root (f(x), x, 0.1, %pi);
(%o10) 1.895494267033981
(%i11) find_root (f, 0.1, %pi);
(%o11) 1.895494267033981
```

A second help manual example is the root of the function $\mathbf{f(x) = e^x - 10}$ which is $\mathbf{x_r} = \ln(\mathbf{10})$.

```
(%i12) f(x) := exp(x) -10$
(%i13) f(0);
(%o13) -9
(%i14) f(100);
(%o14) %e^100-10
(%i15) float(%);
(%o15) 2.6881171418161356E+43
(%i16) f(log(10));
(%o16) 0
```

In the first example below, Maxima does not know a value for **y**, and returns an unevaluated expression (a “noun form” in Maxima-speak). In the second example below, the “variable” in **expr** is **x**, and the value of the parameter **y** is supplied using the pseudo-postfix notation **expr, param = val** .

```
(%i17) find_root (exp(x) = y, x, 0, 100);
(%o17) find_root(%e^x = y,x,0.0,100.0)
(%i18) exact : block([fpprec:20], bfloat(log(10)));
(%o18) 2.302585092994045684b0
(%i19) r2 : find_root (exp(x) = y, x, 0, 100), y = 10;
(%o19) 2.302585092994046
(%i20) block([fpprec:20], bfloat(r2 - exact));
(%o20) 2.1707760372233209909b-16
```

Here is another example of postfix setting of parameters:

```
(%i21) x*cos(y)*exp(z),x=2,y=3,z=4;
(%o21) 2*%e^4*cos(3)
```

To solve the last example with 20 digit precision, one needs to set the *global* value of **fpprec** to the value **20**, and then use **bf_find_root**. The functions **bfloat**, **bf_find_root**, and **bfallroots** (but *not* **float**) compute using the number of digits of precision specified by the current value of **fpprec**. The default value of the global parameter **fpprec** is **16**

```
(%i22) fpprec:20$
(%i23) r3 : bf_find_root (exp(x) = y, x, 0, 100), y = 10;
(%o23) 2.302585092994045684b0
(%i24) exact : block([fpprec:30], bfloat(log(10)));
(%o24) 2.30258509299404568401799145469b0
(%i28) block([fpprec:30], bfloat(r3 - exact));
(%o28) 4.79487947065671528744844106953b-21
```

One can use **fpprec** as a local variable (in a **block** expression) for temporary use of **bfloat**, **bf_find_root**, and **bfallroots**, as in

```
(%i29) fpprec:16;
(%o29) 16
(%i30) block([fpprec:32], bfloat(%pi));
(%o30) 3.1415926535897932384626433832795b0
(%i31) fpprec;
(%o31) 16
```

1.6.7 Maxima Newton-Raphson: newton

If Maxima can symbolically compute the derivative of a given expression (function of a single variable) whose root is sought, then the Newton-Raphson method can find the root if the initial guess is close enough.

The Maxima help manual refers to contributed code in **.../source/numeric/newton1.mac** which has the syntax **newton(expr, var, guess, eps)** which returns the current iteration of **var**, say **v**, if the absolute value of **expr**, when evaluated at **v**, is less than the small positive number **eps**.

Since the Newton-Raphson iteration rule is

$$v^{k+1} = v^k - \frac{f(v^k)}{f'(v^k)}, \tag{1.11}$$

a "divide-by-zero" error will occur if the first derivative is zero at the current value of the iteration variable.

```
(%i1) load("newton1.mac");
(%o1) "C:/PROGRA~1/MA81DB~1.0/share/maxima/5.28.0-2/share/numeric/newton1.mac"
(%i2) newton(x^2 - 5,x,1,1e-4);
(%o2) 2.236068895643363
(%i3) newton(x^2 - 5,x,0,1e-4);
expt: undefined: 0 to a negative exponent.
#0: newton(exp=x^2-5,var=x,x0=0,eps=1.0E-4)(newton1.mac line 8)
 -- an error. To debug this try: debugmode(true);
```

A replacement **newton** function with the same syntax is in the file **cpnewton.mac** on the author's webpage with this chapter and the contents include an exprerimental bigfloat version too.

```
/*  cpnewton.mac  Oct. 2013 */
newton(exp,var,x0,eps):=
block([xn,s,numer,dv],numer:true,
     s:diff(exp,var),
     xn:x0,
     do (if abs(subst(xn,var,exp)) < eps then return(xn),
        dv:subst(xn,var,s),
        if equal(dv,0) then (
           print ("  derivative at",xn,"is zero"),
           return("newton failed")),
        xn:xn-subst(xn,var,exp)/ dv))$
```

```
bfnewton(expr,var,guess,digits):=
block([fpprec, y,eps0,prec,s,xn, bb:2],
       fpprec : digits + bb,
       eps0 : bfloat(10^(-rp)),
       prec : 10^(fpprec/-2.0b0),
       s:diff(expr,var),
       xn:bfloat(guess),
       do (y:subst(xn,var,s),
                   if abs(y) < prec then (
                       print(" derivative at",xn,"is zero"),
                       return(" bfnewton failed")),
               xn : xn - subst(xn,var,expr)/y,
                   xn : rectform(expand(xn)),
               if abs(subst(xn,var,expr)) < eps0 then return(xn)))$
```

A simple example using the version of **`newton`** in **cpnewton.mac**:

```
(%i1) load("cpnewton.mac");
(%o1) "c:/k1/cpnewton.mac"
(%i2) newton(sin(x),x,1.8,1e-5);
(%o2) 6.283185301417648
```

This version of **`newton`** has a graceful exit when the first derivative is zero.

```
(%i3) newton(x^2 - 5,x,1,1e-4);
(%o3) 2.236068895643363
(%i4) newton(x^2 - 5,x,0,1e-4);
  derivative at 0 is zero
(%o4) "newton failed"
```

1.6.8 Maxima Bigfloat Newton-Raphson: bfnewton

The syntax of **`bfnewton`** is **`bfnewton(expr, var, guess, digits)`**, in which **`digits`** is the requested precision of the returned root. The value of **`fpprec`** is set (locally) to a value higher than **`digits`**. There is no need to change the global setting of **`fpprec`**. This is an experimental version of Newton-Raphson.

Here we use **`bfnewton`**, already loaded in with the file **`mynewton.mac`**, to find the positive root of the function **`x^2 - 5`** to 20 digit accuracy.

```
(%i5) fpprec;
(%o5) 16
(%i6) exact : block([fpprec:30],bfloat(sqrt(5)));
(%o6) 2.23606797749978969640917366873b0
(%i7) r20 : bfnewton(x^2 - 5,x,1,20);
(%o7) 2.2360679774997896964096b0
(%i8) block([fpprec:30],r20 - exact);
(%o8) 5.37424113456297828625626672756b-24
(%i9) block([fpprec:30],  subst(r20,x,x^2 -5));
(%o9) 2.40343387953389185011791134874b-23
```

In the last two inputs, using a local setting of **`fpprec`** and **`bfloat`** inside a **`block`** expression, we compared the root found, **`r20`**, with the “exact value” **`exact`**, and evaluated the function at the position of the root found.

Here we check the zero derivative error case:

```
(%i10) bfnewton(x^2 - 5,x,0,20);
 derivative at 0.0b0 is zero
(%o10) " bfnewton failed"
```

1.6.9 Maxima function secant

The secant root finding algorithm is

$$v^{k+1} = v^k - f\left(v^k\right)\frac{v^k - v^{k-1}}{f\left(v^k\right) - f\left(v^{k-1}\right)} \tag{1.12}$$

The secant method requires two initial guesses **x0** and **x1** to get started. The syntax is **secant(f,x0,x1)** or **secant(f,x0,x1,eps)**, in which **f** is either a known function or else a **lambda** form as shown in the second example below.

```
secant(func, vv0, vv1,[oa]) :=
 block([eps0:1e-5,x0,x1 ,x2,ss,jj:0 ,jjmax :3000],
     if length(oa) > 0 then eps0 : oa[1],
     x0 : float(vv0),
     x1 : float(vv1),
     do (jj : jj + 1,
        if jj > jjmax then (
              print(" exceeded jjmax limit "),
              return(x1)),
        if  abs(func(x1)) < eps0 then return(x1),
        ss : func(x1) - func(x0),
        if equal(ss,0) then (
            print(" denominator near",(x0+x1)/2,"is zero "),
            return(x1) ),
        x2 : x1 - func(x1)*(x1 - x0)/ss,
        x0 : x1,
        x1 : x2))$
```

Here is use of a Maxima version of **secant** finding the value of **pi** by finding the root of **sin(x)** near **x = 3**.

```
(%i1) exact : block([fpprec:20],bfloat(%pi));
(%o1) 3.1415926535897932385b0
(%i2) sr : secant(sin,2,3);
(%o2) 3.141592682798589
(%i3) block([fpprec:20],bfloat(sr - exact));
(%o3) 2.9208796148091648731b-8
(%i4) sr : secant(sin,2,3,1e-8);
(%o4) 3.141592653589793
(%i5) block([fpprec:20],bfloat(sr - exact));
(%o5) -1.2246063538223772582b-16
```

and a search for **sqrt(5)**:

```
(%i6) exact : block([fpprec:20],bfloat(sqrt(5)));
(%o6) 2.2360679774997896964b0
(%i7) sr : secant(lambda([x],xˆ2 - 5),1,2);
(%o7) 2.236068111455108
(%i8) block([fpprec:20],bfloat(sr - exact));
(%o8) 1.3395531850186344737b-7
(%i9) sr : secant(lambda([x],xˆ2 - 5),1,2,1e-8);
(%o9) 2.236067977496407
(%i10) block([fpprec:20],bfloat(sr - exact));
(%o10) -3.3825188675939803218b-12
(%i11) sr : secant(lambda([x],xˆ2 - 5),1,2,1e-12);
(%o11) 2.23606797749979
(%i12) block([fpprec:20],bfloat(sr - exact));
(%o12) 1.0864383394662557869b-16
```

1.6.10 Divide and Conquer Root Search

In Computational Physics, Sec. 1.3, Koonin presents a "surefire method" of finding the root of a function when the approximate location of the root is known. (This method cannot find a root in which the function dips down to touch the `x` axis somewhere but doesn't cross the axis.)

You guess a trial value of `x` guaranteed to be less than the root, "increase this trial value by small positive steps, backing up and halving the step size every time the function changes sign." When the value of the step size `dx` is less than some small number, the search returns with the final `x` value found.

Here is a `R` function `rtsearch` which implements this method.

```
rtsearch = function(func,x,dx,xacc) {
    fold = func(x)
    repeat {
      if (abs(dx) <= xacc) break
      x = x + dx
      if (fold*func(x) < 0) {
          x = x - dx
          dx = dx/2}}
    x}
```

After pasting in this definition of `rtsearch` into the `R` Console, we use it to do a brute force search for the positive root of the function $\mathrm{fnf}(x) = x^2 - 5$.

```
fnf = function(x) x^2 -5
> rtsearch(fnf,1,0.5,1e-6)
[1] 2.236067
```

A Maxima version of `rtsearch` is

```
rtsearch(func,xx,dxx,xacc) :=
block([fold,x:xx,dx:dxx],
    fold : func(x),
    do (
      if abs(dx) <= xacc then return(),
      x : x + dx,
      if fold*func(x) < 0 then (
          x : x - dx,
          dx : dx/2)),
    x)$
```

and after pasting this definition into XMaxima, we get

```
(%i2) fnf(x) := x^2 - 5$
(%i3) rtsearch(fnf,1,0.5,1e-6);
(%o3) 2.236066818237305
```

1.7 mtext and Plot Margins in R

Using **?mtext** in **R** brings up (after some editing):

```
mtext {graphics}        R Documentation
Write Text into the Margins of a Plot

Description: Text is written in one of the four margins of the current figure
  region or one of the outer margins of the device region.

Usage:
 mtext(text, side = 3, line = 0, outer = FALSE, at = NA,
           adj = NA, padj = NA, cex = NA, col = NA, font = NA, ...)

Arguments:

1. text:  a character or expression vector specifying the text to be written.
    Other objects are coerced by as.graphicsAnnot.

2. side:  on which side of the plot (1=bottom, 2=left, 3=top, 4=right).

3. line: on which MARgin line, starting at 0 counting outwards.

4. outer: use outer margins if available.

5. at: give location of each string in user coordinates. If the component of at
    corresponding to a particular text item is not a finite value (the default),
        the location will be determined by adj.

6. adj: adjustment for each string in reading direction. For strings parallel
      to the axes, adj = 0 means left or bottom alignment, and adj = 1 means
          right or top alignment.

      If adj is not a finite value (the default), the value of par("las")
          determines the adjustment. For strings plotted parallel to the axis
          the default is to centre the string.

7. padj: adjustment for each string perpendicular to the reading direction
   (which is controlled by adj). For strings parallel to the axes,
   padj = 0 means right or top alignment, and padj = 1 means left or bottom alignment.

   If padj is not a finite value (the default), the value of par("las") determines
   the adjustment. For strings plotted perpendicular to the axis the default is to
   centre the string.

8. cex:  character expansion factor. NULL and NA are equivalent to 1.0. This is an
    absolute measure, not scaled by par("cex") or by setting par("mfrow")
        or par("mfcol"). Can be a vector.

9. col:  color to use. Can be a vector. NA values (the default) mean use par("col").

10. font: font for text. Can be a vector. NA values (the default) mean use par("font").

11 ...          Further graphical parameters (see par), including family, las and xpd.
       (The latter defaults to the figure region unless outer = TRUE, otherwise
              the device region. It can only be increased.)

Details

The user coordinates in the outer margins always range from zero to one, and are not
 affected by the user coordinates in the figure region(s)  R differs here from other
 implementations of S.
```

```
All of the named arguments can be vectors, and recycling will take place to plot as
   many strings as the longest of the vector arguments.

Note that a vector adj has a different meaning from text. adj = 0.5 will centre the
    string, but for outer = TRUE on the device region rather than the plot region.

Parameter las will determine the orientation of the string(s). For strings plotted
         perpendicular to the axis the default justification is to place the end
                 of the string nearest the axis on the specified line. (Note that this
                 differs from S, which uses srt if at is supplied and las if it is not.
                 Parameter srt is ignored in R.)

Note that if the text is to be plotted perpendicular to the axis, adj determines
   the justification of the string and the position along the axis unless at is specified.

Graphics parameter "ylbias" (see par) determines how the text baseline is placed
        relative to the nominal line.

Side Effects

The given text is written onto the current plot.

See Also: title, text, plot, par; plotmath for details on mathematical annotation.
```

You can see the current setting of **`mar`** using **`par()$mar`**. You can see the current setting of **`oma`** using **`par()$oma`**. You can change settings using

```
       par(mar = c(bottom,left,top,right)
       par(oma = c(bottom,left,top,right)
```

The following figure1.R script uses zero outer margin area (the default).

In the **`text`** command, the option **`cex=3`** asks for triple the normal text size.

In the **`mtext`** command, **`side=1`** and **`adj=1`** asks for right justified text in the bottom margin of the figure (use of **`adj=0.5`** would place the text in the bottom center margin), and **`cex=2`** asks for double the default text size. The option **`line=3`** to the **`mtext`** commmand asks that the text be positioned starting at three text lines below the plot region.

```
## figure1.R

plot(0:10, 0:10, type="n", xlab="X", ylab="Y")
text(5,5,"PlotArea", col="red", cex=3)
box("plot", col="red", lwd=2)
mtext("Figure", side=1, line=3, adj=1, cex=2,   col="blue")
box("figure", col="blue",lwd=2)
```

Loading and executing this script in **R** shows that the blue box drawn around the figure does not show up in the **R** Console plot, but *does* show up in the LaTeX/dvi display of **`figure1.eps`**.

```
> source("figure1.R")
```

as shown on the next page.

Here is the resulting plot:

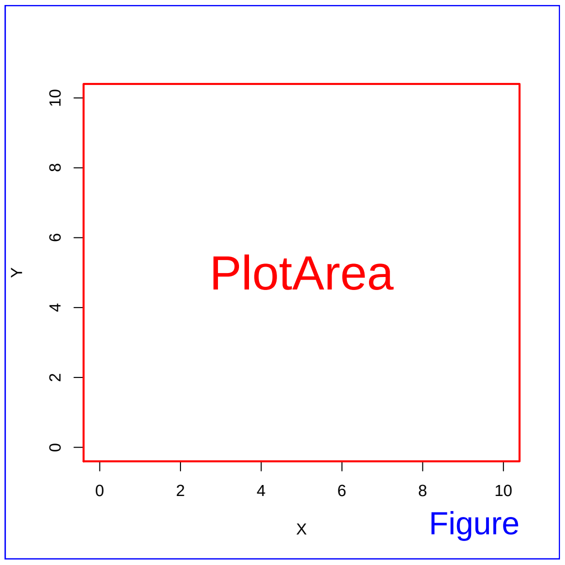

Figure 7: Default No Outer Margin Area

The LaTeX code for this plot was

```
\smallskip
\begin{figure} [h]
   \centerline{\includegraphics[scale=.5]{figure1.eps} }
        \caption{Default No Outer Margin Area}
\end{figure}
```

and makes use of the `graphicx` LaTeX package.

Quoting Earl F. Glynn (Stowers Institute of Medical Research) on the webpage
`http://research.stowers-institute.org/efg/R/Graphics/Basics/mar-oma/`
(with some light editing)

> In Figure 1 the "plot area" is inside the red box, and the perimeter of the figure is shown in a blue box . . . The area between the red box and the blue box is known as the "margin" area, and is controlled by the **R** `mar` parameter. You can view the value of `mar` at any time from the **R** command line: `par()$mar` produces `[1] 5.1 4.1 4.1 2.1`.
>
> You can set the `mar` parameter to other values using the `par` function, as in `par(mar=c(4, 4, 2, 0.5))`. Note that even though the margin is measured in "lines", the values need not be integers.
>
> There is no "outer margin area" in this simple example, which is typical of many **R** plots.

In the next example, an "outer margin area" is added, and messages are placed in that area using `mtext`. The script file figure2.R is

```
## figure2.R
## add outer margin area

oldpar = par(oma=c(0,0,0,0))
par(oma = c(2,2,2,2))
plot(0:10, 0:10, type="n", xlab="X", ylab="Y")
text(5,5,"PlotArea", col="red", cex=3)
box("plot", col="red")
```

```
mtext("Figure", side=1, line=3, adj=1, cex=2,   col="blue")
Margins = " mar = c(5.1, 4.1, 4.1, 2.1) "
mtext(Margins, side=3, line=2, adj=0, cex=1.5, col="blue")
##  "figure" box and "inner" margin box same for single figure plot
box("figure",lty="dashed", col="blue",lwd=2)
box("inner", lty="dotted", col="green",lwd=2)
mtext("Outer Margin Area",side=1, line=0.4, adj=1.0, cex=1.5, col="black", outer=TRUE)
box("outer", lty="solid", col="green",lwd=2)
OuterMargins = " oma = c(2, 2, 2, 2) "
mtext(OuterMargins, side=3, line=0.4, adj=0.0, cex=1.5, col="black", outer=TRUE)
par(oldpar)
```

which produces the figure

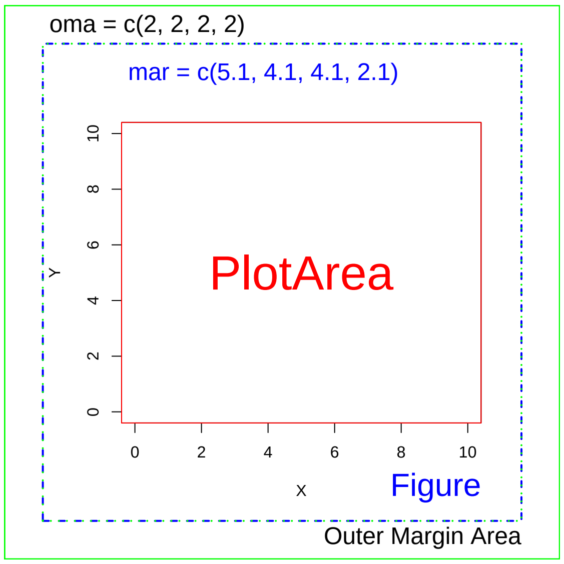

Figure 8: Adding Outer Margin Area

Glynn suggests the replacement of

```
  mtext("Figure", side=1, line=3, adj=1, cex=2,   col="blue")
```

by

```
  mtext("Figure", South <- 1, line=3, adj=1, cex=2,   col="blue")
```

and the replacement of

```
  mtext(Margins, side=3, line=2, adj=0, cex=1.5, col="blue")
```

by

```
   mtext(Margins, North <- 3, line=2, adj=0, cex=1.5, col="blue")
```

which, indeed gives the same result, because the second default arg to **`mtext`** is the value of the option **`side`** (which defaults to 3, implying the top).

This is a case in which one *must* use **`<-`** instead of **`=`**, because **`R`** interprets a function argument of the form **`keyword = value`** differently from the form **`name <- value`**. In the former case, **`R`** finds a keyword it is ready to use and the equal sign is taken to mean: set the keyword to that value. In the latter case, the **`name`** is assigned the value, the name *becomes* a number, and since it appears in the second slot of **`mtext`**, it is taken to be the desired value of the keyword **`side`**.

It is certainly true that, when reading code, **`South <- 1`** is more readily translated mentally to "that is going on the bottom of the figure".

1.8 Drawing Circles with R

1.8.1 Homemade Circles in R

Here is a **`R`** script using **`asp = 1`** inside **`plot`**:

```
##  circle2.R
## homemade circles
tvals = seq(0,2*pi+0.2,0.1)
x = cos(tvals)
y = sin(tvals)
x2 = x/3
y2 = y/3
## note asp=1 y/x setting inside plot
plot(x,y,type = "l",lwd=3,asp = 1)
points(x2,y2,type = "l",lwd=3,col="blue")
abline(h=0,v=0)  # axes
abline(a=0,b=1,col=2,lwd=3)  #red 45 degree line
## perpendiculars from 45 deg line
##  intersections with the circle
segments(cos(pi/4),0,cos(pi/4),sin(pi/4))
segments(0,sin(pi/4),cos(pi/4),sin(pi/4))
segments(-cos(pi/4),0,-cos(pi/4),-sin(pi/4))
segments(0,-sin(pi/4),-cos(pi/4),-sin(pi/4))
```

invoked using **`source`**:

```
> source("c:/k1/circle2.R")
> par()$asp
NULL
```

(see next page for this figure)

Note that **asp** returns to its default setting **NULL** after using **source** with the graphics file **circle2.R**.

Running this script produces:

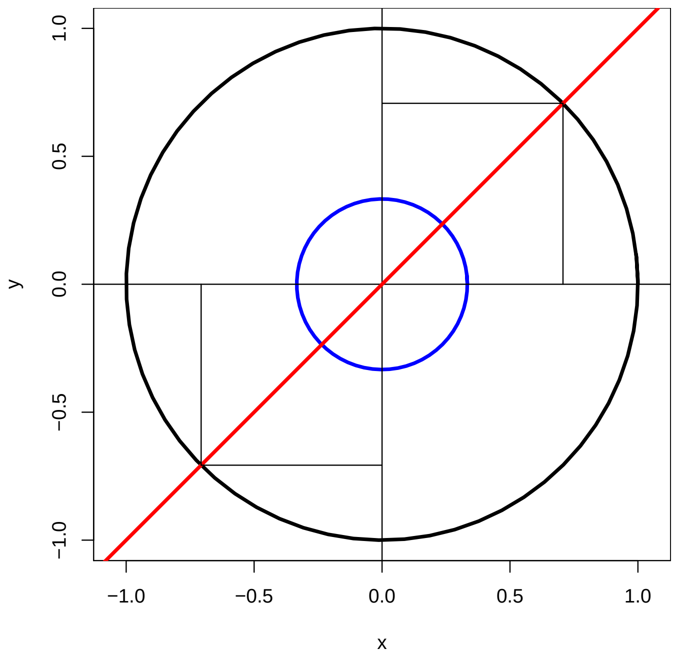

Figure 9: homemade circles

1.8.2 Circles Using the R Package shape

The **shape** package by Karline Soetaert includes functions to create an empty starting plot with **asp = 1** as the default, and to draw ellipses, circles, cylinders, arrows, and rounded rectanges with and without color.

Use

```
library(shape)
help(package = "shape")
vignette("shape")
```

to respectively load the **shape** package, see html documentation of the available functions, and peruse a pdf giving some examples of some of the functions together with the graphic results.

We will restrict ourselves to **emptyplot** and **plotcircle**.

emptyplot has the syntax and defaults:

```
emptyplot(xlim = c(0, 1), ylim = xlim, asp = 1, frame.plot = FALSE,
          col = NULL, ...)
```

which sets up a plot region without axes or frame, with no background color, and with the plotting region extending from **x=0,y=0** to **x=1,y=1**. Thus, using **emptyplot()** without supplying any arguments would accept all these default settings.

plotcircle has the syntax and defaults:

```
plotcircle(r=1,mid=c(0,0),from=-pi,to = pi,type="l", lwd = 2,
                lcol="black", col = NULL, arrow = FALSE,
                arr.length = 0.4, arr.width = arr.length*0.5,
                arr.type = "curved", arr.pos = 1, arr.code = 2,
                arr.adj = 0.5, arr.col = "black",...)
```

with a default radius of one unit, a default center of the circle located at **x=0,y=0**, a full circle is drawn as a line with a slightly thicker style in a black color, with no fill (color) added, and with no arrow added. Note especially that **lcol** determines the line color, and **col** determines the fill color. Thus the two commands:

```
> emptyplot()
> plotcircle()
```

will produce an empty black quarter circle showing only the **x > 0, y > 0** quadrant.

The script

```
## circle3.R
## uses package shape
library(shape)
emptyplot()
plotcircle(mid=c(0.5,0.5),r = 0.1,col = "green",lwd=3)
plotcircle(mid=c(0.5,0.5),r = 0.2,lcol = "blue")
plotcircle(mid=c(0.5,0.5),r = 0.25)
plotcircle(mid=c(0.5,0.5),r = 0.35,lcol = "red",lwd=3)
```

launched with

```
> source("c:/k1/circle3.R")
> par()$asp
NULL
```

produces:

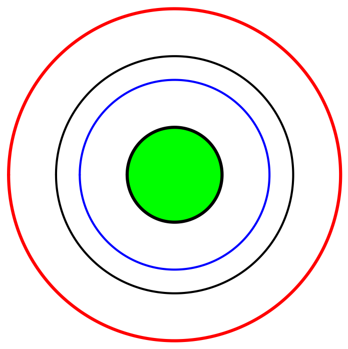

Figure 10: shape package circles

Note that **asp** reverts to its default setting after the script is run.

1.8.3 Circles Using the R Function symbols

The **R** function **symbols** from the package **graphics** is, by default, always available to use, and has the syntax

```
symbols(x, y = NULL, circles, squares, rectangles, stars,
        thermometers, boxplots, inches = TRUE, add = FALSE,
        fg = par("col"), bg = NA,
        xlab = NULL, ylab = NULL, main = NULL,
        xlim = NULL, ylim = NULL, ...)
```

The **R** script **circle4.R**

```
##  circle4.R
##  circles using function symbols

plot(-1:1,-1:1,type="n",asp=1,xlab="",ylab="",bty="n",
        xaxt="n", yaxt="n")

symbols(0,0,circles=1,add=TRUE,lwd=3,inches=FALSE)

symbols(c(0,0,0),c(0,0,0),circles=c(0.75,0.5,0.25),fg = c("green","blue","red"),
       bg=c("white","white","red"),lwd = 3, add = TRUE, inches = FALSE)
```

run with

```
> source("c:/k1/circle4.R")
> par()$asp
NULL
```

produces the plot

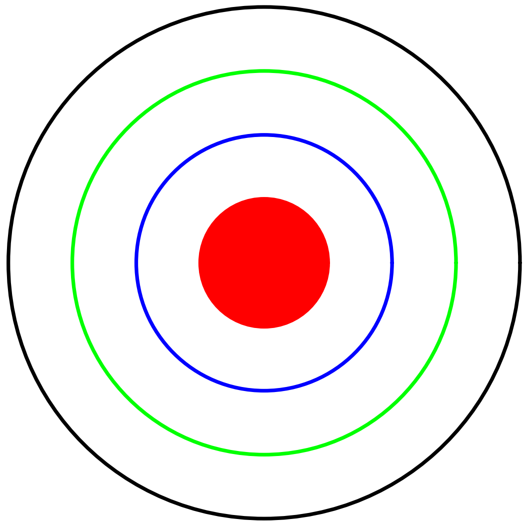

Figure 11: Circles Using symbols(...)

Note that **asp** reverts to its default value after running the script.