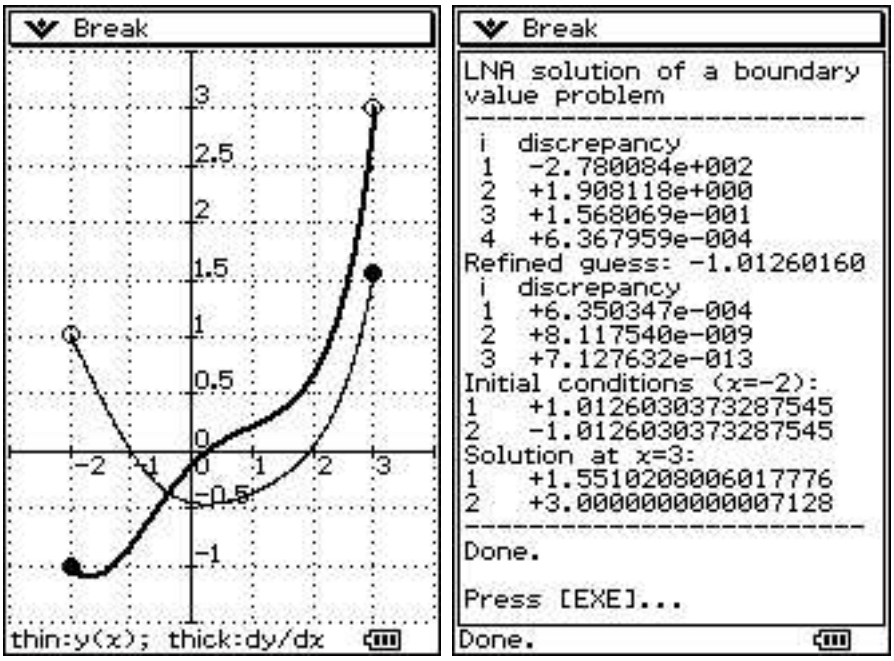
Lua Numerical Analysis (LNA)

A Numerical Analysis package for ClassPad 300

Programmer's manual for version 1.60



Last edited: June 12, 2006.

COVER IMAGES: Solution to the boundary value problem described by the differential equation $\frac{d^2y}{dx^2} = (x+1)\,y + \cos\left(\frac{dy}{dx}\right)$, and the boundary conditions $y(-2) = -\left.\frac{dy}{dx}\right|_{x=-2}$, $\left.\frac{dy}{dx}\right|_{x=3} = 3$. Known boundary conditions (explicitly or implicitly) are marked by non-filled circles. Unknown boundary conditions are marked by filled circles.

Contents

1 Introduction **5**
- 1.1 What is LNA? . . . 5
- 1.2 Disclaimer and Copyright . . . 5
- 1.3 Library organization . . . 5
- 1.4 How to read this manual . . . 6

2 Utility constants and functions **7**
- 2.1 Utility constants . . . 7
  - 2.1.1 Epsilon . . . 7
  - 2.1.2 Pi . . . 7
- 2.2 Utility functions . . . 7
  - 2.2.1 EpsilonC . . . 7
  - 2.2.2 LinSpace . . . 8
  - 2.2.3 MatCol . . . 8
  - 2.2.4 MatIdent . . . 9
  - 2.2.5 MatMul . . . 9
  - 2.2.6 MatPrint . . . 9
  - 2.2.7 MatTrans . . . 10
  - 2.2.8 MaxLoc . . . 10
  - 2.2.9 MinLoc . . . 11
  - 2.2.10 OrderMag . . . 11
  - 2.2.11 Part . . . 12
  - 2.2.12 Nint . . . 12
  - 2.2.13 TimeDiff . . . 12
  - 2.2.14 TimeElapsed . . . 13

3 Plotting functions **15**
- 3.1 Plots of functions . . . 15
  - 3.1.1 PlotFunc . . . 15
- 3.2 Plots of data . . . 19
  - 3.2.1 PlotData . . . 19
- 3.3 Using multiple plotting functions in the same graph . . . 21
- 3.4 "Infinite" plots . . . 23
  - 3.4.1 PlotInf . . . 23

4 Numerical Analysis functions **27**
- 4.1 Root finding . . . 28
  - 4.1.1 Bisect . . . 28
  - 4.1.2 NewtonR . . . 29
  - 4.1.3 Brent . . . 31
  - 4.1.4 KroneRoots . . . 33

4.2 Matrix Algebra . . . 35
4.2.1 LUdecompose . . . 35
4.2.2 LUsubstitute . . . 35
4.2.3 LUsolve . . . 36
4.2.4 LUdeterminant . . . 38
4.2.5 LUinverse . . . 38
4.2.6 Tridiag . . . 40
4.2.7 Jacobian . . . 42
4.2.8 Broyden . . . 44
4.3 Interpolation and extrapolation . . . 46
4.3.1 LinInterp . . . 46
4.3.2 CreateSpline . . . 48
4.3.3 CubicSpline . . . 49
4.3.4 Extrapolation . . . 52
4.4 Curve fitting . . . 52
4.4.1 LMfit . . . 52
4.5 Numerical integration . . . 55
4.5.1 TrapAdapt . . . 55
4.5.2 Romberg . . . 57
4.6 Ordinary differential equations . . . 59
4.6.1 RK4Rich . . . 59
4.6.2 Shoot . . . 62

5 LNA change log **73**
5.1 Version 1.00 (September 19, 2005) . . . 73
5.2 Version 1.10 (September 23, 2005) . . . 73
5.3 Version 1.20 (September 28, 2005) . . . 73
5.4 Version 1.30 (October 5, 2005) . . . 74
5.5 Version 1.40 (October 25, 2005) . . . 74
5.6 Version 1.50 (March 15, 2006) . . . 74
5.7 Version 1.60 (June 12, 2006) . . . 75

6 Bibliography **77**

7 The GNU General Public License **79**

1 Introduction

1.1 What is LNA?

LNA stands for "Lua Numerical Analysis"; it is a package of Numerical Analysis functions, together with several utility and plotting functions. All functions included in the package are meant to be called from a program written in CPLua version 0.8 or above, which runs on a Casio ClassPad 300. This project is continuously developed and it is more than likely that new Numerical Analysis functions, as well as library or plotting functions, will be added in the future. Existing functions may also be modified in a future version, but compatibility with previous versions will be respected, unless it is absolutely necessary to modify the way a function is called.

1.2 Disclaimer and Copyright

1.3 Library organization

The most importrant part of this software is consisting of files organized in three directories: The LNAutils directory, containing utility functions, the LNAplot directory, containing plotting functions, and the LNA directory, containing all Numerical Analysis functions currently included in the package. For each numerical method, there is a file in the directory LNA, containing all functions needed to implement the specific numerical method. In most cases, only one function included in this file can be called by your Lua program (this function will be hereafter referred to as the "main function" — not to be confused with the "main" program). This file usually contains several local (auxiliary) functions needed in the computations, but these functions are hidden to the main program. The file is named as the main function, or an abbreviation of this name, if its length is more than seven characters.

Furthermore, two directories containing examples are included in the package:

1. The LNAexamp directory, which contains one or more simple examples for each numerical method. In order to be easily understandable, the example programs are as simple as possible, and use a limited set CPLua functions. Example programs are therefore a good starting point for the novice user. Each example is named with the letter "X" and the name

of the corresponding function, or an abbreviation of this name, if its length is more than seven characters.

2. The LNAdrive directory, which contains a "driver" program for each numerical method. This driver program covers more complicated examples, and uses more advanced capabilities of the CPLua programming language. Each driver program is named with the letter "D" and the name of the corresponding function, or an abbreviation of this name, if its length is more than seven characters.

For example, consider the LNA implementation of the Romberg method for numerical integration. The main function for this method is named `Romberg`. This function, together with all auxiliary functions needed, is defined in the file Romberg, which is located at the directory LNA. The file XRomberg is a simple program, demonstrating how to use the function `Romberg`, and it is located at the directory LNAexamp. The driver program for the Romberg method is implemented in the file DRomberg, and it is located at the directory LNAdrive.

In addition, a directory named LNAtest is included in the package. This directory contains a "test" program that uses several LNA functions to solve complex test problems. Both the results obtained, and the computation time, are compared with the expected values. LNAtest can be used for testing future versions of LNA or CPLua. As a normal user, you may probably never need this directory.

Almost all Numerical Analysis functions in LNA use other functions (numerical methods and/or utility functions), i.e., they have dependencies. For example, function "A" may use another function, "B", which, in turn, uses the function "C", and so on. This means that, even if your main program calls only one LNA function, you may also need several other functions included in the library. Therefore, *you should not delete or modify any file included in the directories LNAutils, LNAplot and LNA*, unless you know what you are doing.

1.4 How to read this manual

LNA is a package that adds many numerical capabilities in Classpad 300, and includes functions implementing complex algorithms. As such, it is a rather large and complex software. However, using LNA in your CPLua programs is not difficult at all, *provided that you have read and understand the documentation.* Failure to read the documentation may cause you to use LNA in an suboptimal way, or conclude that LNA is useless prematurely.

The best way to get started is to (a) learn how a specific LNA function must be called, and what it returns on exit, (b) study the corresponding example(s) given in this manual. Then you should try to solve another problem, by modifying the example program at will, or by following guidelines included in this manual. Gain experience by solving your own problems: start by a simple problem, write a small program that uses LNA functions to solve it, and check the results. Then try a more difficult problem. Finally, you may want to study the driver program for each numerical method.

Deep knowledge of Numerical Analysis is not necessary. In fact, you can use a specific LNA function to solve practical problems without knowing how the corresponding numerical method works (it is enough to know how it must be used). However, if you are familiar with Numerical Analysis, you may study how each numerical method is implemented in LNA. Remarks for further improving LNA (or this manual) are always welcome.

2 Utility constants and functions

The LNAutils directory includes several utility functions written in Lua. Most of them are called by Numerical Analysis functions included in LNA. Note that, since CPLua is still in development, a future version of CPLua may include built-in utility functions, similar to those presented here. In this case, the corresponding utility function included in the LNAutils directory will be deleted from future versions of LNA.

2.1 Utility constants

2.1.1 Epsilon

The constant `Epsilon` defines the smallest positive number, ϵ, which satisfies the inequality $1+\epsilon > 1$. Due to computer arithmetics, adding a very small number to unity may result *exactly* one. This constant defines `Epsilon` as $\epsilon = 1.12 \times 10^{-16}$.

2.1.2 Pi

The constant `Pi` defines π with the maximum accuracy permitted in ClassPad. Since π is often used in numerical computations, it is defined as a LNA constant, accurate to 16 decimal digits. This constant defines `Pi` as $\pi = 3.1415927410125732$.

2.2 Utility functions

2.2.1 EpsilonC

The function `EpsilonC` computes `Epsilon`, the smallest positive number ϵ, for which $1+\epsilon > 1$.

Syntax

`e=EpsilonC(`*`show,eps,maxit`*`)`

returns ϵ in the varable `e`. All arguments are optional: `show` is a boolean argument that controls whether progress of the calculation will be displayed or not (default: `false`); `eps` is the required accuracy of the result (default: 5×10^{-22}); `maxit` is the maximum number of iterations (default: 100). Usually, there is no need to pass any argument to this function; the default values are sufficient to return an accurate value of `Epsilon`.

Example

`EpsilonC()` returns $1.11022302462516 \times 10^{-16}$, which is the value of `Epsilon` in ClassPad 300, with a maximum error of 5×10^{-22}.

Remarks

`Epsilon` is often used in Numerical Analysis programs as an accuracy tolerance of a numerical method. There is no need to use a very accurate value for this. Therefore, you will not need to use the function `EpsilonC` in your Lua programs. Usually, the value 1.12×10^{-16}, stored in the LNAutils constant `Epsilon`, is more than sufficient.
DEPENDENCIES: None.

2.2.2 LinSpace

The function `LinSpace` returns a vector of equidistant points.

Syntax

```
v=LinSpace(a,b,n)
```

returns a vector `v` of `n` linearly equidistant points between `a` and `b`. This vector contains `n` numbers, representing equally spaced points, between `v[1]=a` and `v[n]=b`.

Examples

1. `LinSpace(0,10,11)` returns `{0,1,2,3,4,5,6,7,8,9,10}`.
2. `LinSpace(-1,1,5)` returns `{-1,-0.5,0,0.5,1}`.
3. `LinSpace(-3,0,4)` returns `{-3,-2,-1,0}`.

Remarks

DEPENDENCIES: None.

2.2.3 MatCol

The function `MatCol` returns a user-specified column of a matrix.

Syntax

```
C=MatCol(A,col)
```

returns a vector `C`, containing the column `col` of a matrix `A`. If the matrix `A` has `n` rows, then `C` is a vector with `n` elements, so that `C[1]=A[1][col]`, `C[2]=A[2][col]`, and so on, until `C[n]=A[n][col]`.

Examples

1. `MatCol({{1,3,2},{4,7,5}},2)` returns `{3,7}`.
2. `MatCol({1,3,2},2)` returns `{3}`.
3. `MatCol({{1},{3},{2}},1)` returns `{1,3,2}`.

Remarks

The result of `MatCol` is always a single-row vector. If `col` is zero or negative, or if `col` is greater than the total number of columns in `A`, the function `MatCol` returns an empty vector, `{}`. If `A` is a single-row vector, `MatCol(A,col)` returns `{A[col]}`, i.e., it returns a vector containing the element `A[col]`.
DEPENDENCIES: None.

2.2.4 MatIdent

The function `MatIdent` returns an identity matrix.

Syntax

```
I=MatIdent(n)
```

returns the identity matrix `I` of rank `n`.

Example

`I=MatIdent(3)` returns `{{1,0,0},{0,1,0},{0,0,1}}`.

Remarks

If `n` is zero or negative, `MatIdent` returns an empty vector, `{}`.
DEPENDENCIES: None.

2.2.5 MatMul

The function `MatMul` implements matrix multiplication.

Syntax

```
C=MatMul(A,B)
```

returns `C=A.B`, if defined. Each argument (`A` or `B`) can be a vector or a matrix.

Examples

1. `MatMul({{1,2},{3,4}},{{5,6},{7,8}})` returns `{{19,22},{43,50}}`.
2. `MatMul({{1,2},{3,4}},{5,6})` returns `{17,39}`.

Remarks

If the argument `A` is a vector, it is considered as a single-row matrix. If the argument `B` is a vector, it is considered as a single-column matrix. If matrix multiplication is not defined for the current arguments, `MatMul` prints an error message, and returns `nil`.
DEPENDENCIES: `MatCol`, `MatTrans`.

2.2.6 MatPrint

The function `MatPrint` prints the elements of a matrix "row by row" (i.e., in a matrix form) using a specific format.

Syntax

`MatPrint(A,`*fmt*`)`

prints iteratively the rows of the matrix `A`. The optional argument `fmt` is a format specifier that defines how each element of the matrix will be printed; if omitted, the value `fmt="%f"` is adopted, which means that each element of `A` will be printed as a float number, rounded to 6 digits.

Examples

1. `MatPrint({3.14159,1.5708,2.71828},"%.2f")` prints: `3.14 1.57 2.72`.
2. `MatPrint({{3.14159,1.5708,0.00318},{2.71828,4,0.00821}},"%.4f")` prints:
   `3.1416 1.5708 0.0032`
   `2.7183 4.0000 0.0082`.

Remarks

If the argument `A` is a vector, it is considered as a single-row matrix. If each row of the matrix does not fit to one line of the console window, it will be printed using more lines, so that the matrix will not be shown in a matrix form.

The format specifier `fmt` follows the C syntax, e.g., "`%i`", "`%.2i`", "`%f`", "`%.2f`", "`%e`", "`%.2e`". See any C or Lua manual for more details on format specifiers.
DEPENDENCIES: None.

2.2.7 MatTrans

The function `MatTrans` transposes a matrix.

Syntax

`B=MatTrans(A)`

returns a matrix `B`, which is `A` transposed.

Examples

1. `MatTrans({1,2,3},{4,5,6})` returns `{{1,4},{2,5},{3,6}}`,
2. `MatTrans({1,2,3})` returns `{{1},{2},{3}}`,
3. `MatTrans({1},{2},{3}})` returns `{1,2,3}`.

Remarks

If `A` is a single-row vector, `MatTrans` returns its elements arranged as a single-column vector. If `A` is a single-column vector, `MatTrans` returns its elements arranged as a single-row vector.
DEPENDENCIES: `MatCol`.

2.2.8 MaxLoc

The function `MaxLoc` locates the position of the maximum element of a vector `A`.

Syntax

```
maxloc,maxval=MaxLoc(A)
```

returns the position of the minimum element, `maxloc`, and its value, `maxval`.

Example

`maxloc,maxval=MaxLoc({1,-1,-5,3,4,2})` returns `5,4`.

Remarks

`MaxLoc` is useful if you want to locate the position of the maximum element. If you just want to compute the maximum value (not its position), you can use `math.max(unpack(A))` instead. DEPENDENCIES: None.

2.2.9 MinLoc

The function `MinLoc` locates the position of the minimum element of a vector `A`.

Syntax

```
minloc,minval=MinLoc(A)
```

returns the position of the minimum element, `minloc`, and its value, `minval`.

Example

`MinLoc({1,-1,-5,3,4,2})` returns `3,-5`.

Remarks

`MinLoc` is useful if you want to locate the position of the minimum element. If you just want to compute the minimum value (not its position), you can use `math.min(unpack(A))` instead. DEPENDENCIES: None.

2.2.10 OrderMag

The function `OrderMag` returns the "order of magnitude" of a number.

Syntax

```
order=OrderMag(x)
```

returns the order of magnitude, `order`, of the number `x`.

Examples

1. `OrderMag(0.0025)` returns `1e-3`.
2. `OrderMag(950)` returns `1e2`.

Remarks

This function is especially useful in plotting functions, for setting the axis and tics scaling.
DEPENDENCIES: None.

2.2.11 Part

The function `Part` returns part of a vector `A`, containing the elements from `A[imin]` to `A[imax]`.

Syntax

```
P=Part(A,imin,imax)
```

returns a vector `P`, with `imax-imin+1` elements, so that `P[1]=A[imin]`, `P[2]=A[imin+1]`, and so on, until `P[imax-imin+1]=A[imax]`.

Example

`Part({1,-1,-5,3,5},2,4)` returns `{-1,-5,3}`.

Remarks

If `imin>imax` the function `Part` returns an empty vector, `{}`.
DEPENDENCIES: None.

2.2.12 Nint

The function `Nint` returns the nearest integer to its argument.

Syntax

```
xi=Nint(x)
```

returns the nearest integer to `x`.

Examples

1. `Nint(1.86)` returns `2`.
2. `Nint(-3.2)` returns `-3`.

Remarks

DEPENDENCIES: None.

2.2.13 TimeDiff

The function `TimeDiff` returns the time passed.

Syntax

```
h,m,s=TimeDiff(hi,mi,si,hf,mf,sf)
```

returns the time passed between initial time `hi`:`mi`:`si` and final time `hf`:`mi`:`sf`. All times are expressed in *hours*:*minutes*:*seconds*.

Examples

1. `TimeDiff(2,5,30,2,10,40)` returns `0,5,40` (time from 2 : 5 : 30 to 2 : 10 : 40 is 0 hours, 5 minutes, 10 seconds).
2. `TimeDiff(21,10,15,9,5,0)` returns `11,54,45` (time from 21 : 10 : 15 to 09 : 05 : 00 is 11 hours, 54 minutes, 45 seconds).

Remarks

`TimeDiff` is primarily used for measuring execution time. As such, it does not check its arguments for validity, since initial and final times are usually taken by calling the built-in function `gettime`. Currently, `TimeDiff` is restricted for measuring time intervals less than one day; the date is not taken into account.
DEPENDENCIES: None.

2.2.14 TimeElapsed

The function `TimeElapsed` returns the time elapsed between a previously recorded time and current time.

Syntax

`h,m,s=TimeElapsed(hi,mi,si,`*print*`)`

returns the time elapsed between initial time `hi:mi:si` and current time. Both times are expressed in *hours*:*minutes*:*seconds*. The optional argument can be anything; if it is present, the time elapsed will be displayed as "Time elapsed: *hours*:*minutes*:*seconds*", otherwise `TimeElapsed` will simply return the time elapsed without printing anything.

Examples

1. `TimeDiff(2,5,30,2,10,40)` returns `0,5,40` (time from 2 : 5 : 30 to 2 : 10 : 40 is 0 hours, 5 minutes, 10 seconds).
2. `TimeDiff(21,10,15,9,5,0)` returns `0,5,40` (time from 21 : 10 : 15 to 09 : 05 : 00 is 11 hours, 54 minutes, 45 seconds).

Remarks

`TimeElapsed` is simply a convenient way for measuring how much time is needed for executing a part of a program; to do this, you should call the functions `gettime` and `TimeElapsed` as follows:

```
hi,mi,si=gettime()
.....................
part of the program
.....................
TimeElapsed(hi,mi,si)
```

FILENAME: TimeElap.
DEPENDENCIES: `TimeDiff`.

3 Plotting functions

The LNAplot directory includes some plotting functions written in CPLua. Currently, plots of 2D functions and/or data are supported. Plots may have tics, tic labels, and grid lines. The user may also select the line thickness, point type, and point size. To achieve this functionality, each plotting function in LNAplot has several optional arguments, and even the mandatory arguments can be used in different ways. It is strongly recommended to study the syntax for each function, and the examples given in this chapter, before using any LNAplot function.

The LNAplot directory contains five files. The file PlotUtil includes several auxiliary plotting utilities, which are required by the user-callable plotting functions; this file should not be loaded directly by any user program. The file DemoPlot is a demonstration program, showing current LNAplot capabilities. The rest of the files in this directory (PlotFunc, PlotData, and PlotInf) can be called by a user program. Their use is described in detail in sections 3.1, 3.2, and 3.3.

It is worth emphasizing that LNAplot is considered as a sub-project of LNA, and, as such, it is developed solely for visualization of the results obtained by the numerical methods included in the package. The author has not the intention to develop a full graphics application; graphical representations not useful for the numerical methods included in LNA will probably never supported by LNAplot.

3.1 Plots of functions

3.1.1 PlotFunc

The function `PlotFunc` plots one ore more user-defined functions.

Syntax

```
axesdata=PlotFunc(f,xv,yv,wait,lwidth,tics,grid,lpos,lsize,c,discont)
```

The first argument, `f`, is a user-defined function (or a table of functions) to be plotted. The second argument, `xv`, is a vector of two elements, describing the x-axis range to be visible. Similarly, the third argument, `yv`, is a vector of two elements, describing the corresponding visible range for the y-axis. The function returns a table `axesdata`, containing information about the axes scaling selected by LNAplot; this output is useful only if the user wants to issue another plotting function to act in the same graph (see section 3.3, page 21 for details). The arguments `wait`, `lwidth`, `tics`, `grid`, `lpos`, `lsize`, `c`, and `discont` are optional. Their meaning is as follows.

1. `wait` is a boolean argument that controls whether the graph window will remain visible after plotting or not (default: `true`). If enabled, the graph will remain visible until you press any key; if disabled, the graph will be generated, but nothing will be displayed on the screen (this is useful if you want to issue another plotting function to act in the same graph).

2. `lwidth` (line width) defines how each function curve will be plotted. `lwidth` can be an integer, or a vector of integers: if it is an integer, all curves will be plotted with a thickness equal to `lwidth` pixels; if it is a vector, each curve will be plotted using the corresponding

thickness number in the vector `lwidth` (the first function curve will be plotted with a thickness equal to `lwidth[1]`, the second with a thickness equal to `lwidth[2]`, and so on). The default value is `lwidth=1`, which means that all function curves will be one-pixel thick. A zero or negative line width means that the corresponding function will not be plotted.

3. `tics` is a vector of two elements: `tics[1]` defines the distance between two consecutive tics in the x-axis, and `tics[2]` defines the corresponding tics distance in the y-axis. Setting any of these values to `"auto"` lets LNAplot decide the tics spacing for the corresponding axis. `tics` can also be set to a single number, instead of a vector of two elements; in this case, both axes will have the same tics spacing. The default value is `tics="auto"` (equivalent to `tics={"auto","auto"}`), which means that LNAplot will automatically select the tics spacing for both axes.

4. `grid` is a boolean argument that controls whether grid lines will be visible or not (default: `true`). Grid lines are shown as dotted vertical and horizontal lines at each x- and y-tic, respectively.

5. `lpos` (label position) defines where the tic labels will be printed. It is a vector of two elements: `lpos[1]` is the x-label position, and `lpos[2]` is the corresponding y-label position. A x-label position equal to 0 means that x-tic labels will be printed at the bottom of the screen; any other value means that x-tic labels will be printed near the corresponding tic of the x-axis. Similarly, a y-label position equal to 0 means that y-tic labels will be printed at the left edge of the screen; any other value means that y-tic labels will be printed near the corresponding tic of the y-axis. `lpos` can also be a number, instead of a vector of two numbers; in this case, x- and y-tic labels will be printed using x-label and y-label position equal to `lpos`. The default value is `lpos={0,0}`, which means that x- and y-tic labels will be printed at the bottom and the left edge of the screen, respectively.

6. `lsize` (label size) defines the size of the tic labels. It is a vector of two elements: `lsize[1]` defines the x-labels size, and `lsize[2]` defines the corresponding y-label size (both in pixels). A zero or negative value for `lsize[1]` or `lsize[2]` means that no labels will be printed for the corresponding axis. `lsize` can also be a number, instead of a vector of two numbers; in this case, x- and y-tic labels will be printed using a label size equal to `lsize`. The default value is `lsize={9,9}`, which means that both x- and y-labels will have a height of 9 pixels.

7. `c` is a vector of two elements, defining where the center of the axes will be. If given, the axes will be crossed at $x =$ `c[1]`, $y =$ `c[2]`; if omitted, the default value is `c="auto"`, which means that the center of the axes will be at $x = 0, y = 0$, as usual. If the center of the axes is outside the visible x- or y-axis range, the corresponding axis will not be visible.

8. `discont` defines the x-coordinates of the discontinuities. It must be a table of the form `{f1disc,f2disc,...}`, where `f1disc` is a vector defining the discontinuities of the first function, `f2disc` is a vector defining the discontinuities of the second function, and so on. For example, `discont={{-1},{0,2}}` means that the first function has a discontinuity at $x = -1$, and the second function has two discontinuities at $x = 0$, and $x = 2$ (the rest of the functions, if present, have no discontinuities). If there is only one discontinuity for a given function, the corresponding argument in `discont` can be a scalar, instead of a vector. For example, `discont={-1,{0,2}}` is equivalent to `discont={{-1},{0,2}}`. The default value is `discont={}`, meaning that all functions have no discontinuities.

Examples

In all the following examples, `f1`, `f2`, `f3`, `f4`, and `f5` are user-defined functions:

1. `PlotFunc(f1,{-1.5,2},{-3,6})` plots the function `f1` for $x \in [-1.5, 2]$. Function values will be visible at the range $y \in [-3, 6]$. This is the most simple use of the `PlotFunc` function, where all optional arguments are set to their default values.

2. `PlotFunc({f1,f2,f3},{-2,2},{-2,3},true,{2,1,1})` plots three functions for $x \in [-2, 2]$. Function values will be visible at the range $y \in [-2, 3]$. The first function curve will be 2-pixels thick, while the others will be 1-pixel thick.

3. `PlotFunc({f1,f2,f3},{-2,2},{-2,3},true,1,{1,0.5},true,1)` plots functions `f1`, `f2`, and `f3` for $x \in [-2, 2]$. Function values will be visible at the range $y \in [-2, 2]$. All function curves will be plotted with a thickness equal to one pixel. Tics in the x-axis will be shown at $x = -2$, -1, 0, 1, 2; tics in the y-axis will be shown at $y = -2$, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2, 2.5, 3. Grid lines will be shown for each x- and y-tic. All labels will be printed near the corresponding tic.

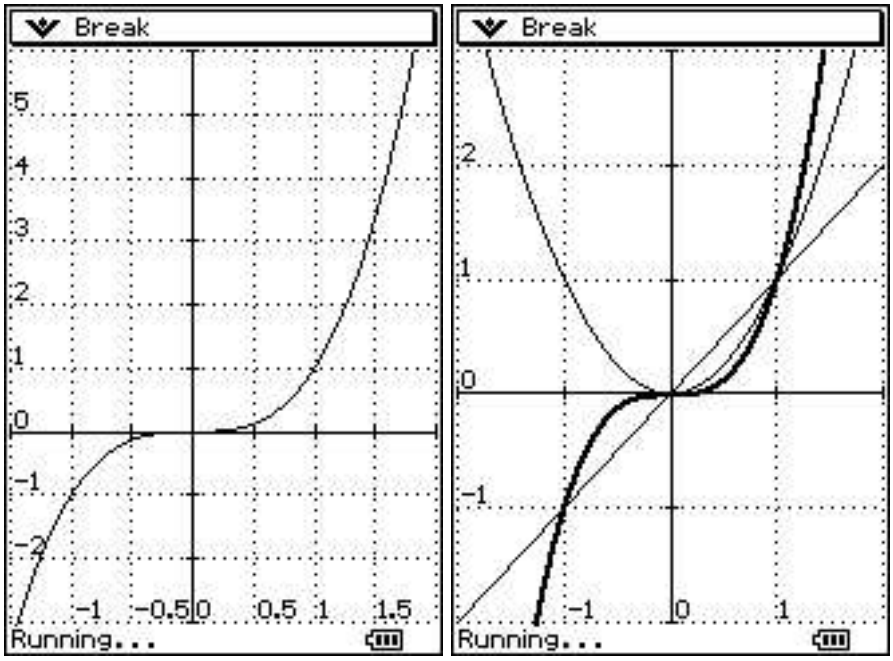
4. `PlotFunc({f3,f4,f5},{-2,6},{-4,4},true,{1,1,2},{2,"auto"},true,1,9,"auto",{nil,nil,2})` plots functions `f3`, `f4`, and `f5` for $x \in [-2, 6]$. Function values will be visible at the range $y \in [-4, 4]$. The last function curve will be 2-pixels thick, while the others will be 1-pixel thick. Tics in the x-axis will be shown at $x = -2$, 0, 2, 4, 6; tic spacing for the y-axis will be computed automatically. Grid lines will be plotted at each tic. All tic labels will be printed near the corresponding tic, and they will have a height of 9 pixels. The axes will be crossed at $x = 0, y = 0$ (automatic selection). The last function curve has a discontinuity at $x = 2$.

These examples are implemented in the example program XPlotFun. Note that `PlotFunc` does not clear the graph window before plotting, so the function `draw.clear()` is called before each call of `PlotFunc`, except the first one.

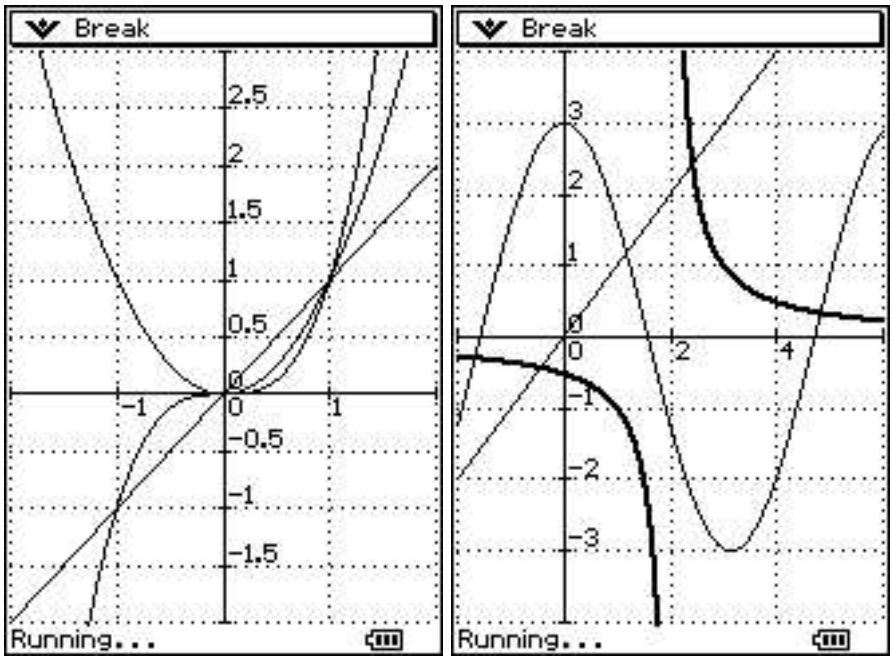
```
require ("draw","LNAplot/PlotFunc")

local function f1(x) return x^3 end
local function f2(x) return x^2 end
local function f3(x) return x end
local function f4(x) return 3*math.cos(x) end
local function f5(x) return 1/(x-2) end

PlotFunc(f1,{-1.5,2},{-3,6})
draw.clear()
PlotFunc({f1,f2,f3},{-2,2},{-2,3},true,{2,1,1})
draw.clear()
PlotFunc({f1,f2,f3},{-2,2},{-2,3},true,1,{1,0.5},true,1)
draw.clear()
PlotFunc({f3,f4,f5},{-2,6},{-4,4},true,{1,1,2},{2,"auto"},true,1,9,"auto"
        ,{nil,nil,2})
```

Example program 1: XPlotFun.

Figure 3.1 shows graphics obtained by running this program.



(a) (b)



(c) (d)

Figure 3.1: Graphics created by the example program XPlotFun.

Remarks

FILENAME: PlotFunc.
DEPENDENCIES: PlotUtil.

3.2 Plots of data

3.2.1 PlotData

The function `PlotData` plots one ore more user-defined data sets.

Syntax

```
axesdata=PlotData(x,y,xv,yv,wait,ptype,psize,lwidth,tics,grid,lpos,lsize,c)
```

The first argument, `x`, is a vector containing the x-coordinate for each data point. The second argument, `y`, is a matrix containing the y-coordinate for each data point; each row of the matrix `y` contains the y-coordinates for each data set; therefore, `y[1]` is a vector containing the y-coordinates for the first data set, `y[2]` contains the y-coordinates for the second data set, and so on; if only one data set is to be plotted, `y` can be a vector (not a matrix). The arguments `xv` and `yv` are equivalent to the corresponding arguments in `PlotFunc`: `xv` is a vector with two elements, describing the visible range in the x-axis; `yv` is a vector with two elements, describing the corresponding visible range for the y-axis. The function returns a table `axesdata`, containing information about the axes scaling selected by LNAplot; this table is useful only if the user wants to issue another plotting function to act in the same graph (see section 3.3, page 21 for details). The arguments `wait`, `ptype`, `psize`, `lwidth`, `tics`, `grid`, `lpos`, `lsize`, and `c` are optional. Their meaning is as follows.

1. `wait` is a boolean argument that controls whether the graph window will remain visible after plotting or not (default: `true`). It is equivalent to the corresponding argument of `PlotFunc`.

2. `ptype` (point type) defines which symbol will be used to mark each data point in each data set. Currently, there are seven point types available in LNAplot: (0) simple point, (1) circle (non-filled), (2) crossed circle, (3) filled circle, (4) rectangular (non-filled), (5) crossed rectangular, and (6) filled rectangular. A point type less than 0 or greater than 6 means that the data points will not be marked. `ptype` can be an integer, or a vector of integers: if it is an integer, data points in all data sets will be marked by a point type equal to `ptype`; if it is a vector, data points in each data set will be marked by the corresponding point type in the vector `ptype` (points in the first data set will be marked by a point type equal to `ptype[1]`, points in the second data set will be marked by a point type equal to `ptype[2]`, and so on). The default value is `ptype=1`, which means that all data points in all data sets will be shown as non-filled circles.

3. `psize` (point size) defines the size of the symbols marking the data points in each data set. `psize` can an integer, or a vector of integers: if it is an integer, data points in all data sets will be marked using a point size equal to `psize`; if it is a vector, data points in each data set will be marked using the corresponding point size in the vector `psize` (points in the first data set will be marked using a point size equal to `psize[1]`, points in the second data set will be marked using a point size equal to `psize[2]`, and so on). The default value is `psize=3`, which means that all points in all data sets will be marked with a point size

equal to 3; for example, if `ptype=1`, and `psize=3` (the default values), data points will be marked by non-filled circles with a radius equal to 3 pixels.

4. `lwidth` (line width) defines how data points in each data set will be joined. `lwidth` can be an integer, or a vector of integers: if it is an integer, data points in all data sets will be joined with line segments having a thickness equal to `lwidth` pixels; if it is a vector, data points in each data set will be joined with line segments having a thickness equal to the corresponding number in the vector `lwidth` (points in the first data set will joined with lines having a thickness equal to `lwidth[1]`, points in the second data set will be joined with lines having a thickness equal to `lwidth[2]`, and so on). The default value is `lwidth=0`, which means that points in all data sets will not be joined by line segments.

5. `tics` defines the distance between two consecutive tics in the x- and y-axis. The default value is `tics="auto"`, which means that the tics spacing for both axes will be automatically selected by LNAplot. It is equivalent to the corresponding argument of `PlotFunc`.

6. `grid` is a boolean argument that controls whether grid lines will be visible or not (default: `true`). It is equivalent to the corresponding argument of `PlotFunc`.

7. `lpos` (label position) defines where the tic labels will be printed. The default value is `lpos={0,0}`, which means that x- and y-tic labels will be printed at the bottom and the left edge of the screen, respectively. It is equivalent to the corresponding argument of `PlotFunc`.

8. `lsize` (label size) defines the size of the tic labels. The default value is `lsize={9,9}`, which means that both x- and y-labels will have a height of 9 pixels. It is equivalent to the corresponding argument of `PlotFunc`.

9. `c` is a vector of two elements, defining where the center of the axes will be. The default value is `c="auto"`, which means that the center of the axes will be at $x = 0, y = 0$. It is equivalent to the corresponding argument of `PlotFunc`.

Examples

In all the following examples, `x`, `y1`, `y2`, `y3`, and `y4` are user-defined vectors:

1. `PlotData(x,y1,{-4,4},{-0.5,1.5})` plots the data set `(x[i],y1[i])` for $x \in [-4, 4]$. The visible y-range will be $y \in [-0.5, 1.5]$. Non-filled circles with a radius equal to 3 pixels will be used to mark each data point. This is the most simple use of the `PlotData` function, where all optional arguments are set to their default values.

2. `PlotData(x,{y1,y2},{-4,4},{-2.5,1.5},true,{2,3},3,{1,0})` plots two data sets for $x \in [-4, 4]$. The visible y-range will be $y \in [-2.5, 1.5]$. Data points in the first data set will be marked as crossed circles, while data points in the second data set will be marked as filled circles. The point size will be 3 pixels for every data set. Points belonging to the first data set will be joined by line segments.

3. `PlotData(x,{y1,y2,y3},{-4,4},{-2.5,1.5},true,{1,3,0},{3,3,0},{1,0,1})` plots three data sets for $x \in [-4, 4]$. The visible y-range will be $y \in [-2.5, 1.5]$. Points in the first data set will be marked as non-filled circles with a radius of 3 pixels, and they will be joined by 1-pixel thick line segments. Points in the second data set will be marked as filled circles with a radius of 3 pixels, but they will not be joined by line segments. Points in the third data set will not be marked, but they will be joined by 1-pixel thick line segments.

4. `PlotData(x,{y3,y4},{-4,4},{-1.5,1.5},true,{1,4},3,1)` plots two data sets for $x \in [-4, 4]$. The visible y-range will be $y \in [-1.5, 1.5]$. Data points in the first data set, `(x[i],y3[i])`, will be marked by non-filled circles, while data points in the second data set, `(x[i],y4[i])`, will be marked by non-filled rectangles. Points in both data sets will be marked using a point size equal to 3 pixels, and they will be joined by line segments.

These examples are implemented in the example program XPlotDat. Since `PlotDat` does not clear the graph window before plotting, the function `draw.clear()` is called before each call of `PlotDat`, except the first one.

```
require ("draw","LNAutils/LinSpace","LNAplot/PlotData")

local x,y1,y2,y3,y4,n

x={};n=15
x=LinSpace(-3.5,3.5,n)
y1={};y2={};y3={};y4={}
for i=1,n do
 y1[i]=x[i]^2/10
 y2[i]=math.sin(x[i])-1
 y3[i]=math.cos(x[i])
 y4[i]=math.sin(math.abs(x[i]))
end
PlotData(x,y1,{-4,4},{-0.5,1.5})
draw.clear()
PlotData(x,{y1,y2},{-4,4},{-2.5,1.5},true,{2,3},3,{1,0})
draw.clear()
PlotData(x,{y1,y2,y3},{-4,4},{-2.5,1.5},true,{1,3,0},{3,3,0},{1,0,1})
draw.clear()
PlotData(x,{y3,y4},{-4,4},{-1.5,1.5},true,{1,4},3,1)
```

Example program 2: XPlotDat.

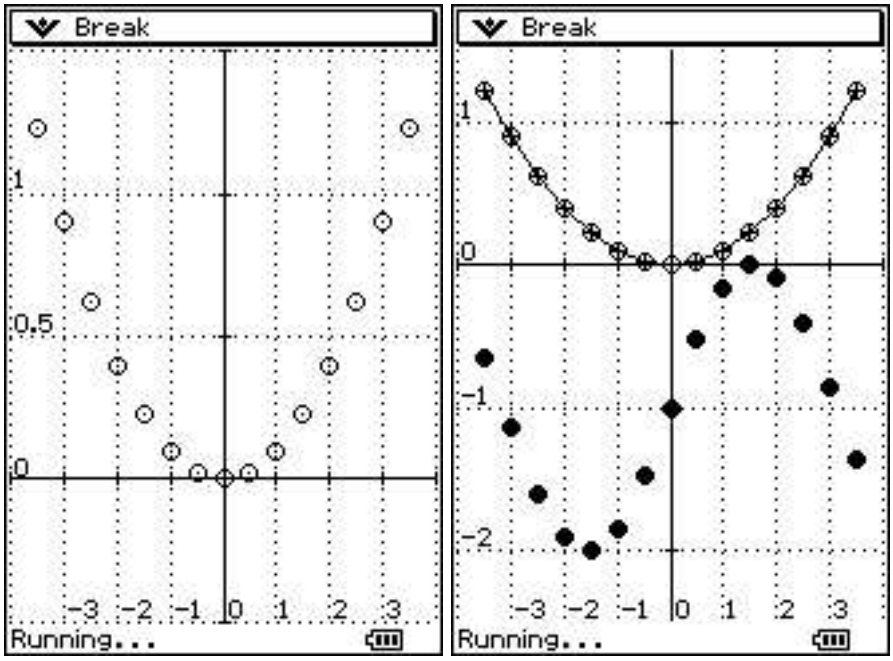
Figure 3.2 shows graphics obtained by running this program.

Remarks

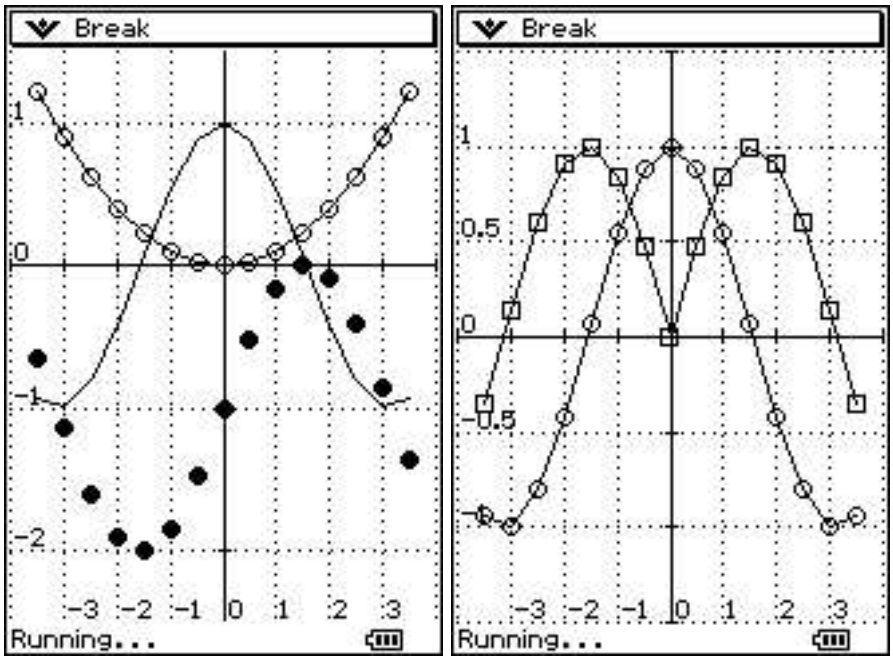
FILENAME: PlotData.
DEPENDENCIES: PlotUtil.

3.3 Using multiple plotting functions in the same graph

Each plotting function can be used to act in a graph already created by another plotting function. To do this, the first plotting function should be called as usual, but the rest plotting functions should be called by setting the argument `xv` equal to `{}`, and the `yv` argument equal to the output of the first plotting function. In other words, `xv={}` is used as a signal that the plotting function should not set any scaling, and `yv` is used to pass the scaling that it is already selected by another plotting function. For example, the following code will plot a function `f` and a set of data points `(x[i],y[i])` in the same graph:



(a) (b)



(c) (d)

Figure 3.2: Graphics created by the example program XPlotDat.

```
axesdata=PlotFunc(f,{-2,2},{-3,3},false)
PlotData(x,y,{},axesdata,true)
```

Note that the first function is called as usual, except that the output is now stored in the variable `axesdata`. This output contains information about the axes scaling selected. In the second function, `xv` is set to `{}`, and `yv` is set to `axesdata`. This informs the second function that it should not set any scaling nor draw axes, and passes the scaling that should be used. Note also that, in the first function, the argument `wait` is set to `false`, so that the second plot will immediately appear after the first. You can, of course, set this argument to `true`, so that the second plot will appear after pressing any key.
IMPORTANT NOTE: To achieve the multiple plots functionality described in this section, all plotting functions do not clear the graph window before plotting. This means that, if you want to create a new graph, you should clear the graph window manually, by using `draw.clear()`.

In the example program of section 3.2, multiple LNAplot functions in the same graph are used to produce figures 3.2b,c,d. In chapter 4, several example programs are using this functionality as well.

3.4 "Infinite" plots

3.4.1 PlotInf

The function `PlotInf` tries to show the behavior of one or more functions $f_1(x), f_2(x), \ldots$ as $x \to \pm\infty$.

Syntax

`PlotInf(f,`*`xr,yr,wait,lwidth,tics,grid,discont`*`)`

The first argument, `f`, is a user-defined function (or a table of functions) to be plotted. The rest of the arguments (`xr`, `yr`, `wait`, `lwidth`, `tics`, `grid`, and `discont`) are optional. `xr` is a vector of two elements, describing the range of x-values. If the first element of `xr` is equal to zero, the minimum x-value will be zero; if it is not, the minimum x-value will be $-\infty$. Similarly, if the second argument of `xr` is equal to zero, the maximum x-value will be zero; if it is not, the maximum x-value will be $+\infty$. In other words, `xr` can be set to (a) `xr={"-inf","inf"}`, i.e. $x \in (-\infty, +\infty)$, (b) `xr={0,"inf"}`, i.e. $x \in [0, +\infty)$, (c) `xr={"-inf",0}`, i.e. $x \in (-\infty, 0]$. Note that `"-inf"` and `"inf"` are used for clarity; any other value, except zero, has the same meaning. The default value is `xr="auto"`, which is equivalent to `xr={"-inf","inf"}` (visible x-range will be $x \in (-\infty, +\infty)$). The argument `yr` is a vector of two elements, describing the range of y-values (default: `yr="auto"`). Valid values for this argument are as in `xr`.

The meaning of the arguments `wait`, `lwidth`, `tics`, `grid`, and `discont` is equivalent to the corresponding arguments of `PlotFunc` (see section 3.1, page 15 for details).

Examples

In all the following examples, `f1`, `f2`, `f3`, `f4`, and `f5` are user-defined functions:

1. `PlotInf({f1,f2,f3})` plots three functions for $x \in (-\infty, +\infty)$. Function values will be visible at the range $y \in (-\infty, +\infty)$. This is the most simple use of the `PlotInf` function, where all optional arguments are set to their default values.

2. `PlotInf({f1,f2,f3},"auto",{0,"inf"})` plots three functions for $x \in (-\infty, +\infty)$. Function values will be visible at the range $y \in [0, +\infty)$.

3. `PlotInf({f1,f2,f4},"auto","auto",true,1,{2,1},true,{nil,nil,{-2,2}})` plots three functions for $x \in (-\infty, +\infty)$. Function values will be visible at the range $y \in (-\infty, +\infty)$. All functions will be plotted with 1-pixel thick curves. Tics in the x-axis will be shown at $x = -2, 2$; tics in the y-axis will be shown at $y = -1, 1$. Grid lines will be shown for each x- and y-tic. Function `f4` is discontinuous at $x = -2$ and $x = 2$.

4. `PlotInf({f1,f2,f3,f5},{0,"inf"},"auto",true,1,"auto",true,{nil,nil,nil,0})` plots four functions for $x \in [0, +\infty)$. Function values will be visible at the range $y \in (-\infty, +\infty)$. All functions will be plotted with 1-pixel thick curves. Tics will be shown at $x = -1, 1$ and $y = -1, 1$. Grid lines will be shown for each x- and y-tic. The last function in the list, `f5`, has a discontinuity at $x = 0$.

The example program XPlotInf implements the examples above. Note that `PlotInf` does not clear the graph window before plotting, so the function `draw.clear()` is called before each call of `PlotInf`, except the first one.

```
require("draw","LNAplot/PlotInf")

local function f1(x) return x end
local function f2(x) return x^2 end
local function f3(x) return math.exp(x) end
local function f4(x) return 5/(x^2-4) end
local function f5(x) return math.log(x) end

PlotInf({f1,f2,f3})
draw.clear()
PlotInf({f1,f2,f3},"auto",{0,"inf"})
draw.clear()
PlotInf({f1,f2,f4},"auto","auto",true,1,{2,1},true,{nil,nil,{-2,2}})
draw.clear()
PlotInf({f1,f2,f3,f5},{0,"inf"},"auto",true,1,"auto",true,{nil,nil,nil,0})
```

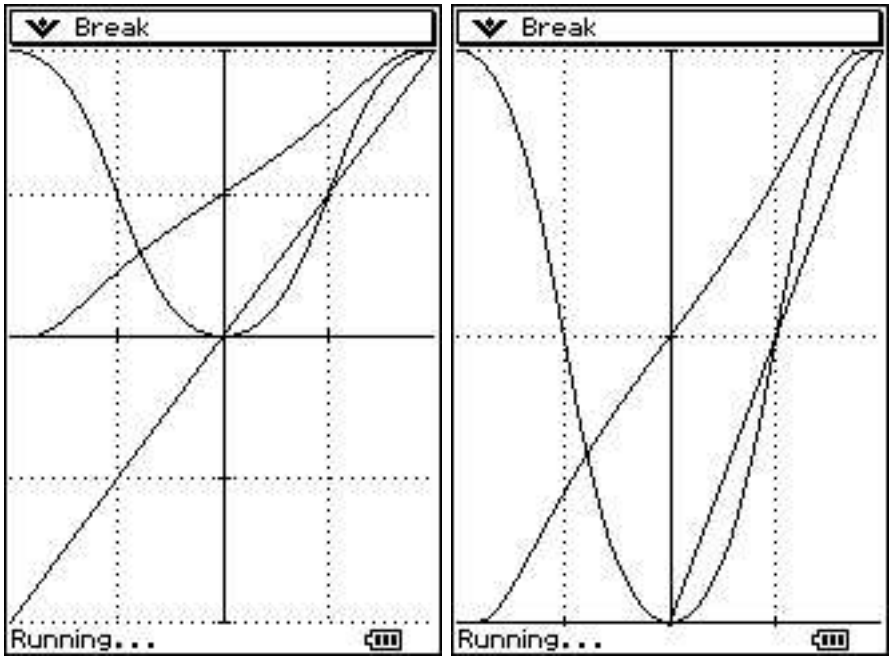
Example program 3: XPlotInf.

Figure 3.3 shows graphics obtained by running this program. Note that several conclusions concerning the limits of the functions can be obtained. For example, in figure 3.3a you can see that $\lim_{x\to-\infty} x = -\infty$, $\lim_{x\to-\infty} x^2 = +\infty$, $\lim_{x\to-\infty} e^x = 0$. In figures 3.3a,b you can see that $\lim_{x\to+\infty} x = +\infty$, $\lim_{x\to+\infty} x^2 = +\infty$, $\lim_{x\to+\infty} e^x = +\infty$, but e^x goes to $+\infty$ faster than x^2, while x^2 goes to $+\infty$ faster than x. Figure 3.3c shows that $\lim_{x\to\pm\infty} \frac{5}{x^2-4} = 0$, $\lim_{x\to-2^-} \frac{5}{x^2-4} = +\infty$, $\lim_{x\to-2^+} \frac{5}{x^2-4} = -\infty$, $\lim_{x\to2^-} \frac{5}{x^2-4} = -\infty$, $\lim_{x\to2^+} \frac{5}{x^2-4} = +\infty$. Figure 3.3d shows that $\lim_{x\to0^+} \ln(x) = -\infty$, and that $\ln(x)$ goes to $+\infty$ slowly, compared to x, x^2, and e^x.

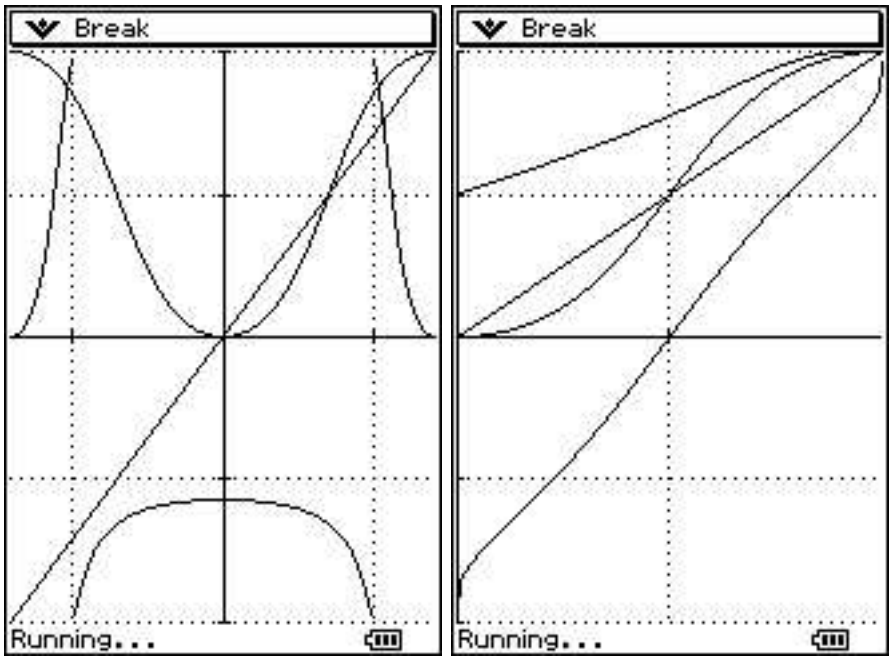
Remarks

"Infinite" plots are useful to inspect the behavior of one or more functions as $x \to \pm\infty$. Scaling in both axes is not linear; if the function to be plotted is oscillating for high $|x|$-values, the details are not shown correctly. In other words, `PlotInf` is not suitable for plotting oscillating functions.

Note that it doesn't make any sense to plot "infinite" plots together with normal plots in the same graph, since scaling is linear in normal plots, while this is not true in "infinite" plots. Therefore, you should not use `PlotInf` together with `PlotFunc` or `PlotData` in the same graph.



(a) (b)



(c) (d)

Figure 3.3: Graphics created by the example program XPlotInf.

FILENAME: PlotInf.
DEPENDENCIES: Pi, PlotUtil, PlotFunc.

4 Numerical Analysis functions

This chapter describes all Numerical Analysis functions currently included in LNA. Although these functions may check some of their arguments for consistency, they are not 100% "idiot proof", since this usually costs computation time. The functions should be considered as "garbage in - garbage out", despite the fact that they usually print warning or error messages if you use them improperly. Reading the documentation, and understanding the example programs before using any LNA function is strongly recommended. The script DemoAll is a menu-driven program that demonstrates all current capabilities of LNA.

You should realize that all functions included in LNA are implementations of numerical methods, and, as such, they may fail to converge, or they may give inaccurate results in special cases. Although the accuracy of the results is controlled by each algorithm, there is nothing "magical" in Numerical Analysis. In general, whenever you use a numerical method, you should accept the fact that there is no numerical method which is able to solve *any* problem, and such a "perfect" method will *never* be. It is always possible that a given numerical method, especially a complex one, may fail in special cases, although this is a rather remote possibility.

Almost all functions described in this chapter have optional arguments, controlling the behavior of the algorithm. In most cases, optional arguments are only useful in special cases. If an optional argument is omitted, it takes a suitable preset value. When describing the syntax of a particular function, optional arguments are written in italic characters.

Most of the functions included in LNA have an optional argument `eps`, which may be used to specify the accuracy of the numerical method. Each function uses specific techniques to estimate the absolute error in the computations, and tries to return a result with an estimated error less than the desired accuracy `eps`. However, you should realize that each numerical method can only make an *estimation* of the error; the actual error may differ, although this difference is usually not important. In special cases, a function may also return a result with no error at all. Be aware that asking for an extremely accurate result (typically, setting an accuracy, `eps`, less than its preset value) may result large computation times, or may lead to an error, due to insufficient memory. A given function may also be unable to return a result as accurate as you asked. Nevertheless, the result may be accurate enough to be useful; in such cases, a warning message is displayed, but the function returns the result obtained (you should, however, check the accuracy of the result). If, on the other hand, the result cannot be useful at all, the function prints an error message, and returns nothing (this usually means that the program execution will be probably stopped).

The rest of this chapter describes the numerical methods currently included in LNA. For each method, a short description and some useful details are given. Obviously, describing in detail how each numerical method works is out of the scope of this manual (see any book on Numerical Analysis for this). Only details concerning how each function can be used in a Lua program are discussed.

4.1 Root finding

4.1.1 Bisect

The function `Bisect` computes the root of a function within a given interval via the Bisection method. This is the simplest (and the slowest) method for root finding. However, its convergence is always guaranteed, and it is often used by more complex numerical methods. The algorithm implemented in `Bisect` is a modification of the "classic" algorithm, sometimes called "Simplified Bisection", and it is slightly faster.

Syntax

```
root,error=Bisect(f,xl,xr,eps,maxit,show)
```

returns the `root` of the function `f` inside the interval `{xl,xr}`, together with an estimation of the absolute `error`. The arguments `eps` , `maxit` and `show` are optional. `eps` defines the desired accuracy (default: `Epsilon`, i.e., 1.12×10^{-16}); it can be set to `"auto"`, which is equivalent to the default accuracy. `maxit` defines the maximum number of iterations (default: 100). `show` is a boolean argument that controls whether progress of the iterative process will be displayed or not (default: `false`); if set to `true`, the function value, `f(root)`, at each iteration will be displayed.

Example

The example program XBisect uses the function `Bisect` to find the root of the function $x - \frac{1}{2}\cos x$ within the interval $[-3, 2]$. In order to check the result, the program computes the function value at the root obtained, and shows the results graphically.

```
require("LNAplot/PlotFunc","LNAplot/PlotData","LNA/Bisect")

local function f(x) return x-math.cos(x)/2 end

local root,error,axesdata

root,error=Bisect(f,-3,2)
print("root:",root)
print("Estimated error:",error)
print("f(root):",f(root))

axesdata=PlotFunc(f,{-3,2},{-2,2},false,1,"auto",true,1)
PlotData({root},{f(root)},{},axesdata,true,2)
```

Example program 4: XBisect.

Figure 4.1 shows the results obtained by running this program. Note that function value, `f(root)`, is exactly zero in this particular example. This is rather the exception than the rule; in most cases, you should expect that `f(root)` will be too close (but not equal) to zero.

Remarks

The interval `{xl,xr}` should contain *exactly* one root; if it does not, or if you need more that one root, you should use the function `KroneRoots` instead.

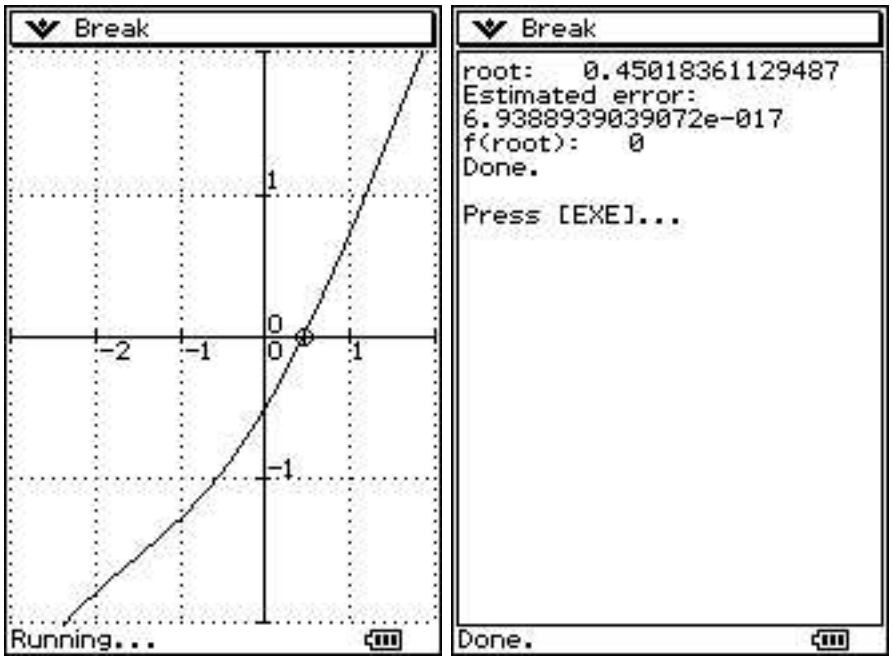


Figure 4.1: Results obtained by the example program XBisect.

In special cases, you may need to reduce the desired accuracy, using the optional argument `eps`. Usually, the maximum number of iterations, `maxit`, does not need to be changed; if the desired accuracy cannot be achieved with the default number of iterations, it is more than likely that the desired accuracy is too high, and cannot be achieved by the Bisection method, despite the number of iterations.

FILENAME: Bisect.

DEPENDENCIES: `Epsilon`.

4.1.2 NewtonR

The function `NewtonR` computes a root of a given function, starting from an initial guess of the root, and using the Newton-Raphson method. This method is widely used due to its convergence speed. However, the method requires the derivative of the function, and may fail to converge, if the initial guess is bad.

Syntax

`root=NewtonR(f,guess,`*`dfdx,eps,maxit,show`*`)`

computes a `root` of the function `f`, starting from an initial guess, `root=guess`. The arguments `dfdx`, `eps`, `maxit`, and `show` are optional. `dfdx` is the name of the function defining the derivative of `f` ; it can be set to `"auto"` (the default), which means that no derivative is supplied (in this case, the derivative of the function will be computed numerically). `eps` defines the desired accuracy (default: `Epsilon`, i.e., 1.12×10^{-16}); it can be set to `"auto"`, which is equivalent to the default accuracy. `maxit` defines the maximum number of iterations (default: 100). `show` is a boolean argument that controls whether progress of the iterative process will be displayed or not (default: `false`); if set to `true`, the function value, `f(root)`, at each iteration will be displayed.

Example

The example program XNewtonR uses the function `NewtonR` to find the root of the function $e^{x^3} - x + 1.5$, starting from the initial guess $x = 0$. The derivative of the function is $-3x^2e^{-x^3} - 1$, and is defined analytically. In order to check the result, the program computes the function value at the root obtained, and shows the results graphically.

```
require("LNAplot/PlotFunc","LNAplot/PlotData","LNA/NewtonR")

local function f(x) return math.exp(-x^3)-x+1.5 end

local function dfdx(x) return -3*x^2*math.exp(-x^3)-1 end

local root,error,axesdata

root,error=NewtonR(f,0,dfdx)
print("root:",root)
print("f(root):",f(root))

axesdata=PlotFunc(f,{-1,3},{-2,4},false,1,"auto",true,1)
PlotData({root},{f(root)},{},axesdata,true,2)
```

Example program 5: XNewtonR.

Figure 4.2 shows the results obtained by running this program. Note that, in this particular example, function value `f(root)` is exactly zero.

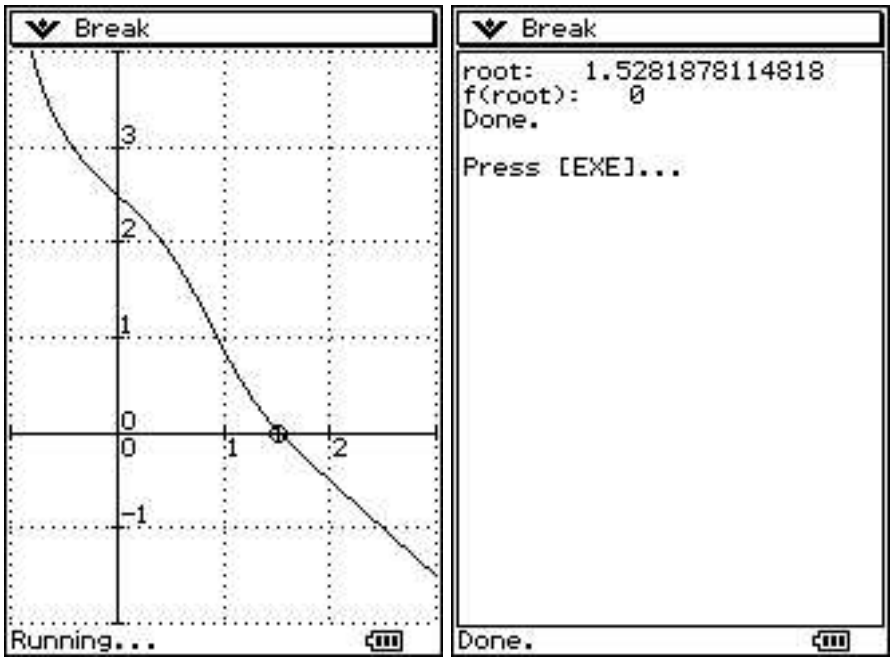


Figure 4.2: Results obtained by the example program XNewtonR.

Remarks

`NewtonR` is able to find one (and only one) root of the function `f`. If there are more than one roots, the root returned depends on the derivative of the function and the initial guess. In other words, `NewtonR` does not necessarily return the root that is close to the initial guess.

If the argument `dfdx` is omitted, `NewtonR` will approximate the derivative numerically. Although this numerical approximation is not trivial, it does not go too far, and may be inaccurate, especially at x close to zero. In general, numerical differentiation is known to be very unsafe, and should be avoided. Consequently, although `dfdx` is an optional argument, you should supply it whenever this is possible.

The initial guess may be crucial for Newton-Raphson method. If the initial guess is bad, the algorithm may fail to converge. As a general rule, select an initial guess for which the derivative is not close to zero. If the method diverges, check the definition of the functions `f` and `dfdx`, or try a different initial guess. Note that the most common error when using `NewtonR` is to define `dfdx` incorrectly; this will cause slow convergence to wrong results.

You may need to reduce the desired accuracy only in special cases. As in `Bisect`, the optional argument `maxit` is rarely needed; if the desired accuracy cannot be achieved with the default number of iterations, it is more than likely that the desired accuracy is too high, or there is an error in the definitions of `f` and `dfdx`.

FILENAME: NewtonR.

DEPENDENCIES: `Epsilon`.

4.1.3 Brent

The function `Brent` computes the root of a function within a given interval via the Brent method. This function combines the guaranteed convergence of the Bisection method and the speed of the Newton-Raphson method (which, however, may diverge). The Brent method is always converging, and it is usually much faster than the Bisection method. Due to these advantages, it is considered as the method of choice for root finding. For convenience, the function `Brent` has the same syntax as `Bisect`.

Syntax

`root=Brent(f,xl,xr,`*`eps,maxit,show`*`)`

returns the `root` of the function `f` inside the interval `{xl,xr}`. The arguments `eps` , `maxit` and `show` are optional. `eps` defines the desired accuracy (default: `Epsilon`, i.e., 1.12×10^{-16}); it can be set to `"auto"`, which is equivalent to the default accuracy. `maxit` defines the maximum number of iterations (default: 100). `show` is a boolean argument that controls whether progress of the iterative process will be displayed or not (default: `false`); if set to `true`, the function value, `f(root)`, at each iteration will be displayed.

Example

The example program XBrent uses the function `Brent` to find the root of the function $e^{1-x} - \frac{x}{2}$ within the interval $[-1, 3]$. In order to check the result, the program computes the function value at the root obtained, and shows the results graphically.

Figure 4.3 shows the results obtained by running this program. Note that function value `f(root)` is too close to zero.

```
require("LNAplot/PlotFunc","LNAplot/PlotData","LNA/Brent")

local function f(x) return math.exp(1-x)-x/2 end

local root,error,axesdata

root,error=Brent(f,-1,3)
print("root:",root)
print("f(root):",f(root))

axesdata=PlotFunc(f,{-1,3},{-2,4},false,1,"auto",true,1)
PlotData({root},{f(root)},{},axesdata,true,2)
```

Example program 6: XBrent.

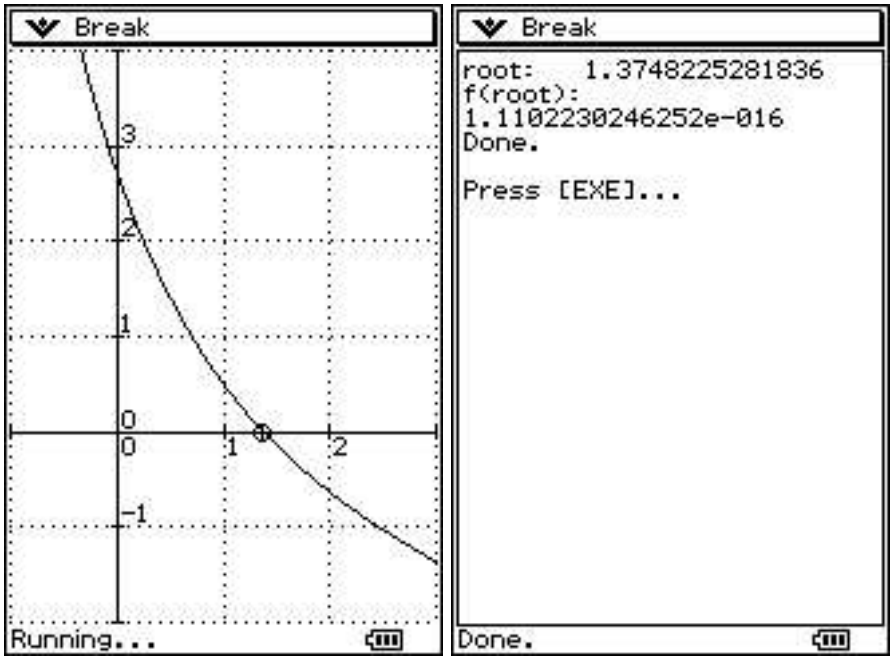


Figure 4.3: Results obtained by the example program XBrent.

Remarks

As in `Bisect`, the interval `{xl,xr}` should contain *exactly* one root. You may need to reduce the desired accuracy only in special cases. The optional argument `maxit` is rarely needed.
FILENAME: Brent.
DEPENDENCIES: `Epsilon`.

4.1.4 KroneRoots

The function `KroneRoots` computes all the roots of a function within a given interval, or, optionally, a prescribed number of roots. It can be used for locating and computing *all* the roots of a function within a given interval. The algorithm implemented in this function is based on the Kronecker-Picard theory (hence its name), and uses recursive auxiliary functions. It is currently the most complex algorithm included in LNA. Despite its complexity, however, the function can be easily called in a user program.

Syntax

`roots,Nr=KroneRoots(f,dfdx,d2fdx2,a,b,`*`r_req,xi`*`)`

returns a vector `roots`, containing all the roots of the function `f` inside the interval `{a,b}`; optionally, it may return only a prescribed number of roots. This function also returns the total number of roots, `Nr`. The arguments `dfdx`, and `d2fdx2` define the first and second derivatives of the function, respectively. The arguments `r_req` and `xi` are optional. The argument `r_req` controls how many roots should be returned. If omitted, or if `r_req<=0`, all the roots will be returned. The optional argument `xi` defines an appropriate number, such that `xi*dfdx(x)` is not too small (or too large) compared to `f(x)`, for all `x` inside the interval `{a,b}`. The default value is `xi=1`.

Example

The example program XKrone uses the function `KroneRoots` to find all the roots of the function $\sin(2x)+\cos(3x)$ within the interval $[-6,2]$. In order to check the result, the program computes the function values at the roots obtained, and shows the results graphically.

Figure 4.4 shows the results obtained by running this program. In this example, there are 8 roots, which are computed accurately.

Remarks

Setting the desired number of roots, `r_req`, may be useful if you only need some roots, but not all of them; in this case the lowest roots will be returned. For example, setting `r_req=1` will return only the lowest root. If `r_req` is greater than the total number of roots, `Nr`, all the roots will be returned, and the user will be informed by a warning message.

The number `xi` is of particular importance for the algorithm. Usually, the default value for `xi` is a good choice. In some cases, however, you may need to change this number in order to obtain all the roots.

The computation time is usually a few seconds, but you should realize that if the function `f` has many roots (and you need all of them), the computation time may become large.
FILENAME: Krone.
DEPENDENCIES: `Bisect`, `Romberg`.

```
require("LNAplot/PlotFunc","LNAplot/PlotData","LNA/Krone")

local function f(x) return math.sin(2*x)+math.cos(3*x) end
local function dfdx(x) return 2*math.cos(2*x)-3*math.sin(3*x) end
local function d2fdx2(x) return -4*math.sin(2*x)-9*math.cos(3*x) end

local Nr,roots,axesdata

roots,Nr=KroneRoots(f,dfdx,d2fdx2,-6,2)
print(Nr.." roots found:")
for i,v in ipairs(roots) do print(i,v) end
froots={}
for j=1,Nr do froots[j]=f(roots[j]) end
print("\nFunction values at roots:")
for i,v in ipairs(froots) do print(i,v) end

axesdata=PlotFunc(f,{-6,2},{-2,2},false)
PlotData(roots,froots,{},axesdata,true,2)
```

Example program 7: XKrone.

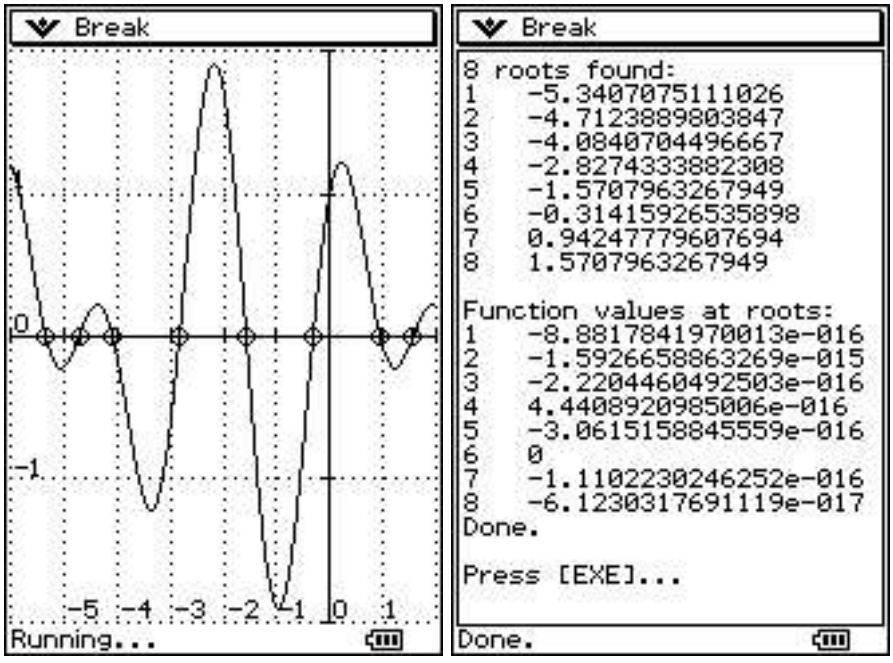


Figure 4.4: Results obtained by the example program XKrone.

4.2 Matrix Algebra

The file LUdecom contains several functions, relative to LU (lower-upper triangular matrix) decomposition. Contrary to most of the numerical methods included in LNA, the functions included in this file are all available to the user.

4.2.1 LUdecompose

The function `LUdecompose` decomposes a matrix to its LU equivalent. LU decomposition of a matrix `A` is a method aiming to find a lower-triangular matrix, `L`, and an upper-triangular matrix, `U`, so that `A=L.U`. Such a decomposition is used mainly for solving a system of linear equations, but can also be used to compute the determinant or the inverse of a matrix. The method implemented in `LUdecompose` performs LU decomposition with partial pivoting.

Syntax

```
LU,indx,parity=LUdecompose(A)
```

returns a matrix `LU`, which contains the lower- and upper-triangular equivalents of the input matrix `A`. It also returns a vector `indx`, which records the row permutations effected by the algorithm, and a number `parity`, equal ± 1, depending on whether the number of row interchanges was even or odd, respectively.

Remarks

The function `LUdecompose` is called as a first step to solve a system of linear equations, and/or compute the determinant or the inverse of a matrix. You will never need to call this function without calling one or more of the functions `LUsubstitute`, `LUdeterminant`, and `LUinverse`, described below.

Note that, if the input matrix `A` is singular, the function prints a warning message, and returns `nil`.

FILENAME: LUdecom.

DEPENDENCIES: None.

4.2.2 LUsubstitute

The function `LUsubstitute` is used after calling `LUdecompose` in order to solve a system of linear equations. It performs "forward substitution" and "backward substitution", in order to solve a system of linear equations `A.X=B`, where the matrix `A` has been previously decomposed to its LU equivalent.

Syntax

```
X=LUsubstitute(LU,indx,B)
```

returns a vector `X`, which is the solution of the system of linear equations `A.X=B`. The first argument is the LU equivalent of the matrix `A`, and the second is a vector `indx`, which records the row permutations effected by LU decomposition (these arguments are taken from the output of `LUdecompose`). The third argument, `B`, is a vector containing the right-hand side of the equations to be solved.

Remarks

The function `LUdecompose` *must* be called before calling `LUsubstitute`. In other words, the commands

```
LU,indx,parity=LUdecompose(A)
X=LUsubstitute(LU,indx,B)
```

are used to solve a system of linear equations `A.X=B`.

Note that `LUsubstitute` returns `nil`, if there is no solution, i.e., if the matrix `A` is singular.

FILENAME: LUdecom.

DEPENDENCIES: None.

4.2.3 LUsolve

The function `LUsolve` solves a system of linear equations. It is simply a "shortcut" function. It calls `LUdecompose` and `LUsubstitute` internally, in order to solve a system of linear equations.

Syntax

```
X=LUsolve(A,B)
```

returns a vector `X`, which is the solution of the system of linear equations `A.X=B`.

Example

The example program XLUsolve uses the function `LUsolve` to find the solution of the system of linear equations

$$\begin{aligned} x + 2z &= 9 \\ -x + 4y + 3z + 6w &= 12 \\ -2y + 5z - 3w &= 3 \\ 3x + y + z &= 15. \end{aligned}$$

In order to check the result, the program also computes the product `A.X`, by using the utility function `MatMul` (see section 2.2.5, page 9 for details).

Figure 4.5 shows the results obtained by running this program. Note that `A.X` is found to be equal to the vector `B`, as required.

Remarks

This function is useful if you only want to solve a system of linear equations; it solves the system using LU decomposition, hiding the details from the user. However, if you also want to compute the determinant and/or the inverse of the matrix `A`, it is preferable to solve the system by successively calling `LUdecompose` and `LUsubstitute`. If you call `LUsolve`, but you also want to compute, e.g., the determinant of the matrix `A`, you will need to call `LUdecompose` again, thus doing unnecessary computations.

Note that `LUsolve` returns `nil`, if there is no solution, i.e., if the matrix `A` is singular.

FILENAME: LUdecom.

DEPENDENCIES: Depends on `LUdecompose` and `LUsubstitute`, also included in the file LUdecom.

```
require("LNAutils/MatMul","LNAutils/MatPrint","LNA/LUdecom")

local A,B,X

A={{1,0,2,0},{-1,4,3,6},{0,-2,5,-3},{3,1,1,0}}
B={9,12,3,15}

print("Coefficients A:")
MatPrint(A,"%+.1f")
print("\nConstants B:")
print(unpack(B))

X=LUsolve(A,B)
print("\nSolution X:")
print(unpack(X))
print("\nSolution verification, A.X:")
print(unpack(MatMul(A,X)))
```

Example program 8: XLUsolve.

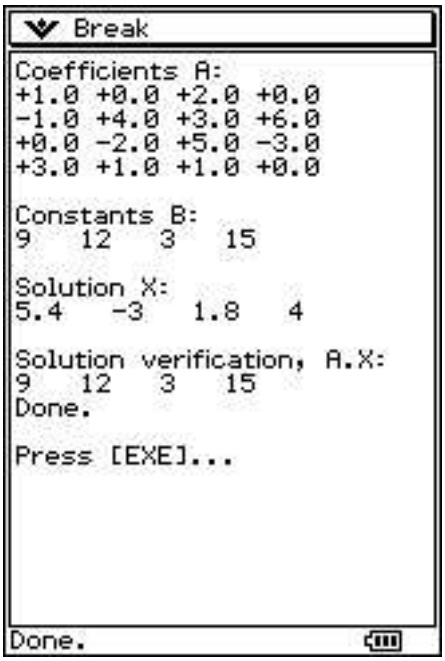


Figure 4.5: Results obtained by the example program XLUsolve.

4.2.4 LUdeterminant

The function `LUdeterminant` is used after calling `LUdecompose`, and computes the determinant of a matrix `A`, which has been previously decomposed to its LU equivalent.

Syntax

```
detA=LUdeterminant(LU,parity)
```

returns the determinant, `detA`, of a matrix `A`. `LU` is the LU equivalent of the matrix `A`, and `parity` is a number, equal ± 1, depending on whether the number of row permutations was even or odd, respectively. Both arguments are taken from the output of `LUdecompose`.

Remarks

The function `LUdecompose` *must* be called before calling `LUdeterminant`. In other words, the commands

```
LU,indx,parity=LUdecompose(A)
detA=LUdeterminant(LU,parity)
```

are used to compute the determinant of the matrix `A`.
FILENAME: LUdecom.
DEPENDENCIES: None.

4.2.5 LUinverse

The function `LUinverse` computes the inverse of a matrix `A`, which has been previously decomposed to its LU equivalent. It is used after calling `LUdecompose` in order to compute the inverse of a matrix.

Syntax

```
Ainv=LUinverse(LU,indx)
```

returns a matrix `Ainv`, which is the inverse of a matrix `A`. The first argument is the LU equivalent of the matrix `A`, and the second is a vector `indx`, which records the row permutations effected by LU decomposition. Both arguments are taken from the output of `LUdecompose`.

Example

The example program XLUdecom uses the function `LUdecompose` to decompose the same system of linear equations, as in section 4.2.3, page 36. Then the functions `LUsubstitute`, `LUdeterminant`, and `LUinverse` are used to compute the solution, the determinant, and the inverse of matrix `A`, respectively.

Figure 4.6 shows the results obtained by running this program. Note that the solution `X` obtained by calling `LUdecompose` and `LUsubstitute` is equivalent to the solution obtained by calling `LUsolve` directly (cf. figure 4.5, page 37).

```
require("LNAutils/MatPrint","LNA/LUdecom")

local A,B,X,LU,indx,parity,detA,Ainv

A={{1,0,2,0},{-1,4,3,6},{0,-2,5,-3},{3,1,1,0}}
B={9,12,3,15}

print("Coefficients A:")
MatPrint(A,"%+.1f")
print("\nConstants B:")
print(unpack(B))

LU,indx,parity=LUdecompose(A)

X=LUsubstitute(LU,indx,B)
print("\nSolution X:")
print(unpack(X))

detA=LUdeterminant(LU,parity)
print("\nDeterminant detA="..detA)

Ainv=LUinverse(LU,indx)
print("\nInverse matrix, Ainv:")
MatPrint(Ainv,"%+.2f")
```

Example program 9: XLUdecom.

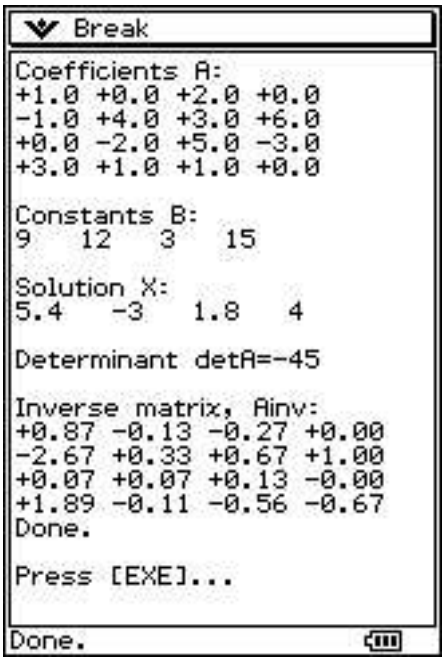


Figure 4.6: Results obtained by the example program XLUdecom.

Remarks

The function `LUdecompose` *must* be called before calling `LUinverse`. In other words, the commands

```
LU,indx,parity=LUdecompose(A)
Ainv=LUinverse(LU,indx)
```

are used to compute the inverse of the matrix `A`.

Note that `LUinverse` returns `nil`, if the inverse matrix is not defined.

FILENAME: LUdecom.

DEPENDENCIES: `MatCol`, `MatIdent`, `MatTrans`.

4.2.6 Tridiag

The function `Tridiag` solves a tridiagonal system of linear equations. It implements LU decomposition for the special case of tridiagonal systems.

Syntax

```
X=Tridiag(a,d,c,B)
```

returns a vector `X`, which is the solution of the system of linear equations `A.X=B`, where the matrix `A` is tridiagonal; the sub-diagonal vector is `a`, the diagonal vector is `d`, and the above-diagonal vector is `c`. For a tridiagonal system `A.X=B` of N equations, the arguments `a` and `d` must have N elements, while the argument `c` must have $N-1$ elements. The first element of the sub-diagonal vector `a` is ignored (can be `nil`) and the rest elements of `a` define the coefficients `A[2,1]`, `A[3,2]`, `A[4,3]`, ..., `A[`N`,`$N-1$`]`. The diagonal vector `d` defines the coefficients `A[1,1]`, `A[2,2]`, `A[3,3]`,..., `A[`N`,`N`]`. The above-diagonal vector `c` defines the coefficients `A[1,2]`, `A[2,3]`, `A[3,4]`,..., `A[`$N-1$`,`N`]`. The rest of the elements of the coefficient matrix `A` are assumed to be zero.

Example

The example program XTridiag uses the function `Tridiag` to solve the system of linear equations

$$\begin{aligned} 3x - z &= 4 \\ x + 4y + 2z &= -7 \\ 3y + 5z - w &= -15 \\ -2z + 7w &= 18. \end{aligned}$$

In this case, the coefficient matrix A is tridiagonal:

$$A = \begin{bmatrix} 3 & -1 & 0 & 0 \\ 1 & 4 & 2 & 0 \\ 0 & 3 & 5 & -1 \\ 0 & 0 & -2 & 7 \end{bmatrix}.$$

Figure 4.7 shows the results obtained by running this program.

```
require("LNA/Tridiag")

local a,d,c,b,x

a={nil,1,3,-2}
d={3,4,5,7}
c={-1,2,-1}
b={4,-7,-15,18}
x=Tridiag(a,d,c,b)

print("Sub-diagonal a:")
print(unpack(a))
print("\nDiagonal d:")
print(unpack(d))
print("\nAbove-diagonal c:")
print(unpack(c))
print("\nConstants b:")
print(unpack(b))
print("\nSolution x:")
print(unpack(x))
```

Example program 10: XTridiag.

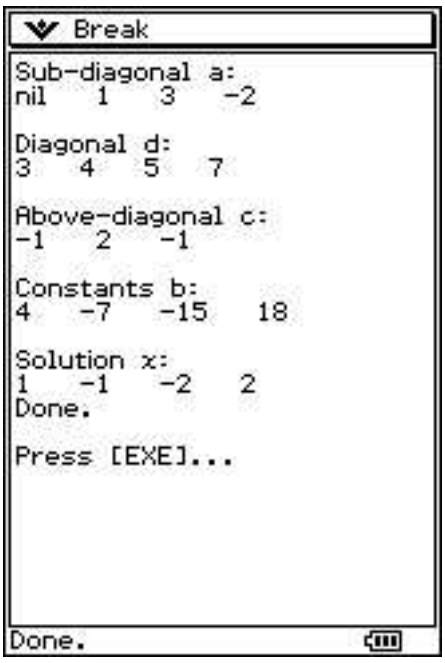


Figure 4.7: Results obtained by the example program XTridiag.

Remarks

A tridiagonal system `A.X=B` can, of course, be solved by `LUsolve` (see section 4.2.3, page 36). However, using `Tridiag` instead of `LUsolve` saves both memory and computation time. By using `Tridiag`, only the non-zero elements of the matrix `A` must be defined.
FILENAME: Tridiag.
DEPENDENCIES: None.

4.2.7 Jacobian

The function `Jacobian` computes the Jacobian of a multivariate function numerically.

Syntax

```
J=Jacobian(F,x)
```

returns a matrix `J`, which is the Jacobian of a multivariate function `F`. This function must be defined as `F(x)`, where `x` is a vector defining the independent variables $x_1, x_2, \ldots, x_N$; it must return a vector of N elements:

$$F(x) = F(x_1, x_2, \ldots, x_N) = \begin{bmatrix} F_1(x_1, x_2, \ldots, x_N) \\ F_2(x_1, x_2, \ldots, x_N) \\ \vdots \\ F_N(x_1, x_2, \ldots, x_N) \end{bmatrix}.$$

The result obtained is the matrix

$$J(x) = \begin{bmatrix} \frac{\partial F_1}{\partial x_1} & \frac{\partial F_1}{\partial x_2} & \cdots & \frac{\partial F_1}{\partial x_N} \\ \frac{\partial F_2}{\partial x_1} & \frac{\partial F_2}{\partial x_2} & \cdots & \frac{\partial F_2}{\partial x_N} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial F_N}{\partial x_1} & \frac{\partial F_N}{\partial x_2} & \cdots & \frac{\partial F_N}{\partial x_N} \end{bmatrix}$$

at the specified values `x={x[1],x[2],...,x[N]}`.

Example

The example program XJacobia uses the function `Jacobian` to compute the Jacobian of the function

$$F(x) = F(x_1, x_2, x_3) = \begin{bmatrix} x_1^2 - 2x_2 + x_3 - 2 \\ x_1^3 - 4x_3^2 + 7 \\ x_1 + x_2 x_3^2 - 1 \end{bmatrix}.$$

at the point $(1, 2, -1)$.

Figure 4.8 shows the results obtained by running this program. Note that the Jacobian can be computed analytically:

$$J(x_1, x_2, x_3) = \begin{bmatrix} 2x_1 & -2 & 1 \\ 3x_1^2 & 0 & -8x_3 \\ 1 & x_3^2 & 2x_2 x_3 \end{bmatrix}.$$

The result obtained numerically is very accurate, and almost equal to the correct value, $J(1, 2, -1)$. In this case, the maximum absolute error is 10^{-8}.

```
require("LNAutils/MatPrint","LNA/Jacobia")

local function F(x)
return {x[1]^2-2*x[2]+x[3]-2,x[1]^3-4*x[3]^2+7,x[1]+x[2]*x[3]^2-1}
end

local Japprox

Japprox=Jacobian(F,{1,2,-1})
print("Aproximate Jacobian:")
MatPrint(Japprox,"%+.4f")
```

Example program 11: XJacobia.

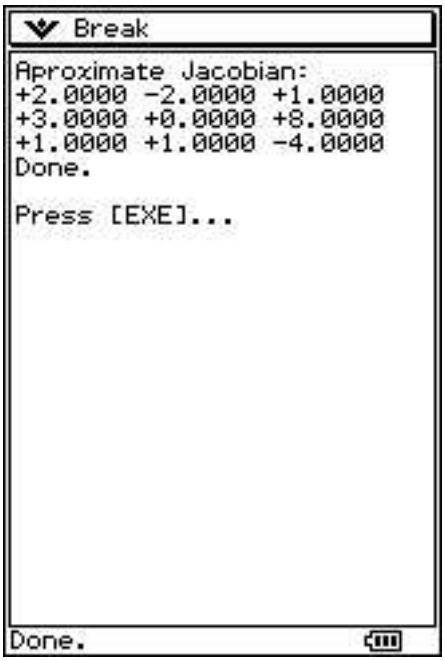


Figure 4.8: Results obtained by the example program XJacobia.

Remarks

Although numerical approximation of the Jacobian is usually accurate enough, it is always better (and always possible) to define the Jacobian analytically.
FILENAME: Jacobia.
DEPENDENCIES: None.

4.2.8 Broyden

The function `Broyden` solves a system of non-linear equations by implementing Broyden's method. This method is iterative, and tries to find a solution, starting by a user-supplied initial guess. It is often considered as the method of choice for solving systems of non-linear equations.

Syntax

`root=Broyden(xguess,F,J,eps,maxit,show)`

returns a vector `root`, which is the solution of a system of non-linear equations. The general form of a system of N non-linear equations with N unknowns $x_1, x_2, \ldots, x_N$ is $F(x) = 0$, which is expressed analytically as

$$\begin{aligned} F_1(x_1, x_2, \ldots, x_N) &= 0 \\ F_2(x_1, x_2, \ldots, x_N) &= 0 \\ &\vdots \quad \vdots \quad \vdots \\ F_N(x_1, x_2, \ldots, x_N) &= 0. \end{aligned}$$

`xguess` is the initial guess of the solution, and should be a vector of N elements. `F` is the name of a user-defined function `F(x)`, which returns a vector defining the left-hand-side of the equations to be solved, given a vector `x` defining the independent variables. The arguments `J`, `eps`, `maxit`, and `show` are optional.

The argument `J` must be the name of a function `J(x)` defining the Jacobian matrix $J(x)$. If omitted, this argument takes the default value `J="auto"`, which means that the Jacobian will be computed numerically.

`eps` defines the desired accuracy (default: `Epsilon`, i.e., 1.12×10^{-16}); `maxit` defines the maximum number of iterations (default: 20); `show` is a boolean argument that controls whether progress of the calculation will be displayed or not (default: `false`).

Example

Consider the system of non-linear equations

$$\begin{aligned} x_1 - x_2^4 + x_3 - 5 &= 0 \\ x_1^2 + x_2^3 - 3 &= 0 \\ x_1 + x_2 x_3^2 &= 0. \end{aligned}$$

In this case, the multivariate function $F(x)$ is defined as

$$F(x) = F(x_1, x_2, x_3) = \begin{bmatrix} x_1 - x_2^4 + x_3 - 5 \\ x_1^2 + x_2^3 - 3 \\ x_1 + x_2 x_3^2 \end{bmatrix}.$$

Its Jacobian is

$$J(x) = J(x_1, x_2, x_3) = \begin{bmatrix} 1 & -4x_2^3 & 1 \\ 2x_1 & 3x_2^2 & 0 \\ 1 & x_3^2 & 2x_1x_2 \end{bmatrix}.$$

The example program XBroyden uses the function `Broyden` to find the solution of this system of non-linear equations with initial guess $x_1 = 2$, $x_2 = 0$, $x_3 = 3$. In order to check the result, the program computes the function values at the solution obtained.

```
require("LNA/Broyden")

local function F(x)
return {x[1]-x[2]^4+x[3]-5,x[1]^2+x[2]^3-3,x[1]+x[2]*x[3]^2}
end

local function J(x)
return {{1,-4*x[2]^3,1},{2*x[1],3*x[2]^2,0},{1,x[3]^2,2*x[2]*x[3]}}
end

local root

root=Broyden({2,0,3},F,J)
print("Solution:")
for i,v in ipairs(root) do print(i,v) end
print("\nFunction value:")
for i,v in ipairs(F(root)) do print(i,v) end
```

Example program 12: XBroyden.

Figure 4.9 shows the results obtained by running this program. Note that, in this example, the Jacobian is defined by the user. Alternatively, one can call the function `Broyden` as `root=Broyden({2,0,3},F)`. In this case, the Jacobian is omitted, and it is computed numerically.

The reader should modify the initial guess in the above example, to see how the solution obtained is affected by a "bad" initial guess. For example, try the following cases:

1. `xguess={3,1,6}` is not a good initial guess, but the solution obtained is correct.
2. `xguess={1,0,5}` is not a good initial guess, but the solution obtained is accurate, although the maximum number of iteration is reached, and the accuracy criterion is not satisfied.
3. `xguess={0,1,0}` is a very bad initial guess; the maximum number of iteration is reached, the accuracy criterion is not satisfied, and the "solution" obtained is wrong.

Remarks

A good choice of the initial guess `xguess` is crucial for Broyden's algorithm. Although you may get correct results using an initial guess far from the correct solution, it is always better to supply a good initial guess. Failure to do so may cause `Broyden` to return a totally wrong "solution"; in this case, a warning message will be printed, and you should alter the initial guess.

Usually, there is no need to change the default value for the "accuracy" `eps`. The algorithm should converge after a few iterations; if it does not, it is almost sure that it will never converge,

Figure 4.9: Results obtained by the example program XBroyden.

due to wrong initial guess, or wrong input data. In other words, you will probably never need to use the optional argument `maxit`. The most common error when using `Broyden` is to define `F` or `J` incorrectly; in most cases, this will cause wrong results, obtained after the maximum number of iterations is reached.

If the Jacobian `J` is omitted, it is computed numerically by using the function `Jacobian` (see section 4.2.7, page 42). Although numerical computation of the Jacobian is usually accurate enough, it is better to supply it by using the argument `J`. In most cases, computing the Jacobian analytically is not difficult at all.

FILENAME: Broyden.

DEPENDENCIES: `Epsilon`, `MatMul`, `MatTrans`, `Jacobian`, `LUdecom`.

4.3 Interpolation and extrapolation

4.3.1 LinInterp

The function `LinInterp` performs linear interpolation (or extrapolation).

Syntax

```
v=LinInterp(xpoint,dernumber,xdata,ydata,warn ser)
```

returns the value of the linear interpolating function (or its first derivative) at a specified point `xpoint`. The argument `dernumber` is an integer defining the order of the derivative to be computed. Setting `dernumber=0` means that the value of the interpolating function will be computed; setting `dernumber=1` will return the first derivative at $x =$ `xpoint`. The arguments `xdata` and `ydata` must be two vectors, containing the x- and y-coordinates for each data point, respectively.

If `xpoint` lies outside the data range, extrapolation will be performed, and the result may not be accurate. The optional boolean argument `warnuser` controls whether the user should be warned in this case (default: `true`).

Example

In the example program XLinInt, a set of 19 data points concerning the function $f(x) = \sin(x)$ is defined, within the interval $[-2, 7]$. Then, interpolating functions approximating $f(x)$ and $\frac{df}{dx}$ are defined by using `LinInterp`. The results are shown graphically. For simplicity, data points are equally spaced, i.e., the vector `xdata` is created by using `LinSpace` (see section 2.2.2, page 8).

```
require("draw","LNAplot/PlotData","LNAplot/PlotFunc","LNAutils/LinSpace"
       ,"LNA/LinInt")

local xdata,ydata,axesdata

xdata=LinSpace(-2,7,19)
ydata={}
for i=1,#xdata do
 ydata[i]=math.sin(xdata[i])
end
local function flinear(x)
return LinInterp(x,0,xdata,ydata)
end
local function dflinear(x)
return LinInterp(x,1,xdata,ydata)
end

axesdata=PlotData(xdata,ydata,{-2,7},{-1,1},false,1,3,0,"auto",true,1)
PlotFunc(flinear,{},axesdata)
draw.clear()
PlotFunc(dflinear,{-2,7},{-1,1},true,2,"auto",true,1)
```

Example program 13: XLinInt.

Figure 4.10 shows the results obtained by running this program. In figure 4.10a, the data points are plotted, together with the interpolating function approximating $f(x)$. Figure 4.10b shows the interpolating function approximating $\frac{df}{dx}$.

In this example, $f(x)$ is well approximated by linear interpolation (the error is rather small). The approximated derivative $\frac{df}{dx}$ is a piecewise constant function, and it is obviously not accurate.

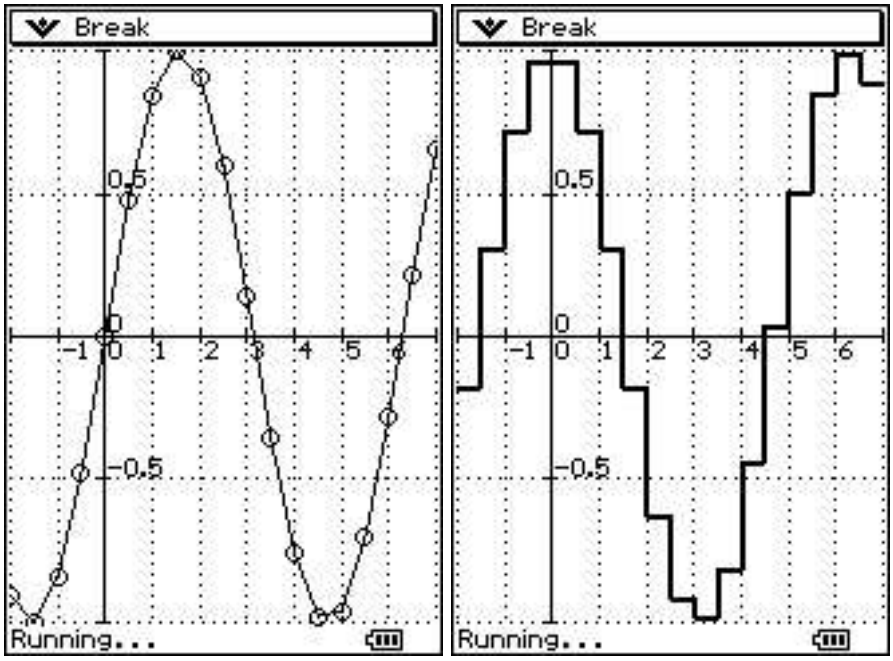
The reader should modify the number of data points to see how the accuracy of the results is affected. For example, changing the number of data points to 55 (instead of 19) will result a much more accurate approximation for the first derivative.

Remarks

A sufficiently large set of data points is crucial for obtaining accurate results using linear interpolation. Note that linear interpolation produces a piecewise linear function; therefore, its derivative is by nature a piecewise constant function. Obviously, the interpolating function is not differentiable at all data points.

The vector `xdata` must contain monotonically increasing or decreasing data. If this is not the case, you must sort the elements of vector `xdata` (making also the appropriate changes in vector `ydata`) before calling `LinInterp`. Note that x-coordinates stored in the vector `xdata` need not to be equally spaced.

FILENAME: LinInt.



(a) (b)

Figure 4.10: Results obtained by the example program XLinInt.

DEPENDENCIES: None.

4.3.2 CreateSpline

The function `CreateSpline` computes all data needed to define a cubic spline interpolating function.

Syntax

`scoeff=CreateSpline(xdata,ydata,`*`stype,endder`*`)`

returns the spline coefficient data, `scoeff`, needed to perform cubic spline interpolation. The arguments `xdata` and `ydata` must be two vectors, containing the x- and y-coordinates for each data point, respectively. The arguments `stype` and `endder` are optional. The argument `stype` is a string, defining the type of the spline function to be created. It must have one of the following values:

Spline type	Meaning
"not-a-knot"	The third derivative of the spline function is assumed to be continuous at the data points adjacent to the end points.
"clamped"	The first derivative at the end points is known. In this case, the optional argument `endder` must be present.
"prolonged"	The curve is "prolonged", so that end data points are treated as internal points.
"natural"	The well-known "natural" cubic spline. The second derivative at the end points is assumed to be zero.
"quadratic"	The second derivative is assumed to be constant near the end points.

The default value is `stype="not-a-knot"`, which is usually the best choice, if you don't have any additional information concerning the underlying function.

`endder` is a vector of two elements, defining the first derivative at the end points. `endder[1]` must be equal to the first derivative at $x =$ `xdata[1]`; similarly, `endder[2]` must be equal to the first derivative at $x =$ `xdata[N]`, where `N` is the total number of data points. Note that `endder` must be supplied only if `stype` is defined as "clamped"; in all other cases, this argument is ignored.

Remarks

From an accuracy point of view, use essentially the "clamped" spline type, if you know the end point derivatives, otherwise use "not-a-knot". The "prolonged" spline type is sometimes used as an alternative to "not-a-knot". Despite its name, the "natural" spline type is not a good choice, unless you know that the second derivative of the underlying function is equal to zero at the end points; if you don't have any reason to assume this, do not use "natural" splines. In a similar manner, the "quadratic" spline type should only be used if you know that the second derivative is almost constant near the end points.

As in, `LinInterp`, the vector `xdata` must contain monotonically increasing or decreasing data. However, x-coordinates stored in this vector need not to be equally spaced.

FILENAME: CSpline.

DEPENDENCIES: `Part`, `Tridiag`.

4.3.3 CubicSpline

The function `CubicSpline` creates a cubic spline interpolating function using spline coefficient data previously computed by `CreateSpline`.

Syntax

```
v=CubicSpline(xpoint,dernumber,xdata,scoeff,warn ser)
```

returns the value of the cubic spline interpolating function (or its first or second derivative) at a specified point `xpoint`. The argument `dernumber` is an integer defining the order of the derivative to be computed. Setting `dernumber=0` means that the value of the interpolating function will be computed; setting `dernumber=1` or `dernumber=2` will return the first or the second derivative, respectively, at $x =$ `xpoint`. As in `CreateSpline`, the argument `xdata` is a vector containing the x-coordinates for each data point. The argument `scoeff` is a matrix containing cubic spline coefficient data, as computed by `CreateSpline`.

The optional boolean argument `warnuser` has the same meaning as in `LinInterp` (see section 4.3.1). It controls whether the user should be warned if `xpoint` lies outside the data range (default: `true`).

Example

The example program XCSpline is similar to XLinInt. A set of 10 data points concerning the function $f(x) = \sin(x)$ is defined, within the interval $[-2, 7]$. The function `CreateSpline` is used to compute spline data; then, cubic spline functions approximating $f(x)$, $\frac{df}{dx}$, and $\frac{d^2f}{dx^2}$ are defined by using `CubicSpline`. The results are shown graphically. For simplicity, the vector `xdata` is created by using `LinSpace`, so data points are equally spaced, although this is not necessary.

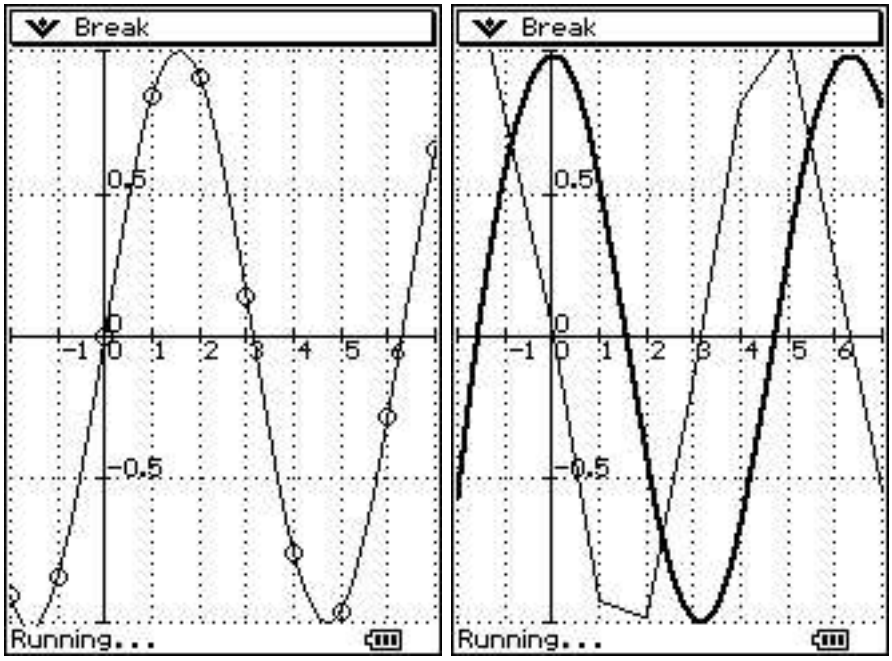
```
require("draw","LNAplot/PlotData","LNAplot/PlotFunc","LNAutils/LinSpace"
       ,"LNA/CSpline")

local xdata,ydata,scoeff,axesdata

xdata=LinSpace(-2,7,10)
ydata={}
for i=1,#xdata do
 ydata[i]=math.sin(xdata[i])
end
scoeff=CreateSpline(xdata,ydata)
local function fspline(x)
return CubicSpline(x,0,xdata,scoeff)
end
local function dfspline(x)
return CubicSpline(x,1,xdata,scoeff)
end
local function d2fspline(x)
return CubicSpline(x,2,xdata,scoeff)
end

axesdata=PlotData(xdata,ydata,{-2,7},{-1,1},false,1,3,0,"auto",true,1)
PlotFunc(fspline,{},axesdata)
draw.clear()
PlotFunc({dfspline,d2fspline},{-2,7},{-1,1},true,{2,1},"auto",true,1)
```

Example program 14: XCSpline.

Figure 4.11 shows the results obtained by running this program. In figure 4.11a, the data points are plotted, together with the spline function approximating $f(x)$. Figure 4.11b shows the spline functions approximating $\frac{df}{dx}$ (thick line) and $\frac{d^2f}{dx^2}$ (thin line).

In this example, $f(x)$ is very well approximated by cubic splines (the error is very small). The approximated derivative $\frac{df}{dx}$ has a larger error, but it is still accurate. On the other hand, the error in approximated $\frac{d^2f}{d^2x}$ is considerably larger. This is the expected behavior of cubic splines.

A simple comparison of figures 4.11 and 4.10 shows what should be expected if cubic spline interpolation is used, instead of linear interpolation. It is obvious that the first derivative is much more accurately approximated by cubic splines, although fewer data points were used.



(a) (b)

Figure 4.11: Results obtained by the example program XCSpline.

The reader should modify the number of data points to see how the accuracy of the results is affected. For example, changing the number of data points to 30 (instead of 10) will result very accurate approximations, even for the second derivative. Note that, in this example, the default (`"not-a-knot"`) spline type is used. The reader should try different spline types. For example, the command

```
scoeff=CreateSpline(xdata,ydata,"clamped",{math.cos(-2),math.cos(7)})
```

can be used instead of `scoeff=CreateSpline(xdata,ydata)`. This way, a more accurate "clamped" spline function will be created.

Remarks

Before calling `CubicSpline`, the function `CreateSpline` must be called. It is important to understand that `CreateSpline` must be called *only once* to compute the necessary spline data. Once this has been done, values of the interpolating function (or its derivatives) can be obtained by calling `CubicSpline` as many times as necessary.

Cubic splines are often useful to compute the first or the second derivative numerically. Although numerical differentiation is known to be very unsafe, estimating the derivatives by cubic spline interpolation is usually accurate, provided that the underlying function is sufficiently smooth, and a sufficient amount of data points is available.

In general, cubic spline interpolation is considered as the method of choice, especially in situations where continuity of the first derivative is a concern.

FILENAME: CSpline.

DEPENDENCIES: `Part`, `Tridiag`.

4.3.4 Extrapolation

The functions `LinInterp` and `CubicSpline` can be used to *extrapolate* data, i.e., to compute the value of the function (or its derivatives) outside the data interval. In this case, a warning message will be printed, to inform the user that extrapolation has been requested. This warning message can be switched off by setting the optional argument `warnuser` to `false` (this is useful if you want to avoid multiple warning messages printed by `LinInterp` or `CubicSpline`). To see how extrapolation works, modify the example programs XLinInt and XCSpline, so that the interpolating function will be plotted for $x \in [-3, 8]$. Since data points in these example programs are available in the range $x \in [-2, 7]$, extrapolation will be used for $x \in [-3, -2]$ and $x \in [7, 8]$.

It is worth emphasizing that extrapolation is *very* unsafe, and should be avoided. In rare cases, however, extrapolation could be useful. Even if you need to perform extrapolation, do so for x-values near the end points. Setting `xpoint` outside the data interval, and too far from the end data points, will probably return a totally erroneous result.

4.4 Curve fitting

4.4.1 LMfit

The function `LMfit` performs a non-linear fit of data to a theoretical function, using the Levenberg-Marquardt algorithm. It finds a set of parameters $a = a_1, a_2, \ldots, a_m$, such that a set of data (x_i, y_i) $i = 1, 2, \ldots, n$ is best fitted by a function $f(x, a) = (x, a_1, a_2, \ldots, a_m)$. To do this, the function implements the Levenberg-Marquardt algorithm, which is a rather complex numerical method, aiming to find a set of parameters a, such that the well-known quantity χ^2 ("chi-square") is minimized. Despite its complexity, however, `LMfit` can be easily used in a Lua program, in order to find a theoretical curve that best models a set of data.

Syntax

```
chi2,a,Da=LMfit(f,dfda,aguess,xdata,ydata,show,sigma,eps,maxit)
```

finds a vector containing a set of parameters, `a`, so that the data set `(xdata[i],ydata[i])` is best fitted by a function `f(x,a)`. `f` is a user-defined theoretical function that models the data; it depends on `x` and the set of parameters, `a`. `dfda` is a function defining the partial derivatives of `f` with respect to the parameters `a`; this function should return a vector containing the derivatives $\frac{\partial f}{\partial a_1}, \frac{\partial f}{\partial a_2}, \ldots, \frac{\partial f}{\partial a_n}$. `aguess` is a vector containing the initial guess for all the parameters `a`. `xdata` and `ydata` are two vectors of the same size, containing the x- and y-coordinates of the data points. The arguments `show`, `sigma`, `eps` and `maxit` are optional. `show` is a boolean argument that controls whether progress of the calculation will be displayed or not (default: `true`). `sigma` defines the standard deviation of the data points; it can be a number or a vector of the same size as `xdata` and `ydata`: if it is a number, all data points are considered to have a standard deviation equal to `sigma`; if it is a vector, each data point `(xdata[i],ydata[i])` has a standard deviation equal to `sigma[i]`; if `sigma` is omitted, all data points are taken with a standard deviation equal to 1. The optional argument `eps` defines the desired accuracy of the algorithm (default: `eps=0.01`), and the argument `maxit` defines the maximum number of iterations (default: 20). The function returns the final (minimized) value for χ^2, `chi2`, and the estimated value for all the parameters, `a`. If the specified standard deviation is constant to all data points, `LMfit` also returns a vector `Da`, containing the estimated confidence interval for all the parameters (with a risk equal to 5%). In other words, the final value for the first parameter is `a[1]`$\pm$`Da[1]`, the final value for the second parameter is `a[2]`$\pm$`Da[2]`, and so on.

Example

Consider a hypothetical experiment, where a physical quantity y is measured, as a function of another quantity x. It is expected that y relates to x according to $y = \sin(4x + 1) + 2\cos(2x)$. The following data set has been obtained by the experiment:

x	y
-1.5	-1.0
-1.0	-1.0
-0.5	$+0.3$
$+0.0$	$+2.8$
$+0.5$	$+1.1$
$+1.0$	-1.8
$+1.7$	-1.0

We want to find a curve of the form

$$f_{\text{fit}}(x) = \sin(a_1 x + a_2) + 2\cos(a_3 x)$$

that best fits this data. According to theory, if the measurements are accurate enough, we must find $a_1 = 4$, $a_2 = 1$, $a_3 = 2$. However, actual parameters obtained may vary, due to errors in measurements. The example program XLMfit uses the function `LMfit` to find the optimal values of the parameters a_1, a_2, a_3, starting from the initial guess $a_1 = 3$, $a_2 = 0$, $a_3 = 1$. The program also uses LNAplot functions to plot the data set and the fitting curve. Note that, in this case, the partial derivatives $\frac{\partial f}{\partial a_1}$, $\frac{\partial f}{\partial a_2}$, and $\frac{\partial f}{\partial a_3}$ are given by

$$\begin{aligned} \frac{\partial f}{\partial a_1} &= x\cos(a_1 x + a_2), \\ \frac{\partial f}{\partial a_2} &= \cos(a_1 x + a_2), \\ \frac{\partial f}{\partial a_3} &= -2x\cos(a_3 x), \end{aligned}$$

and the user-supplied function `dfda` is defined accordingly.

Figure 4.12 shows the results obtained by running this program. We find $a_1 = 4.04 \pm 0.11$, $a_2 = 1.07 \pm 0.07$, and $a_3 = 2.00 \pm 0.04$. Note that `LMfit` was able to find very accurate values for the parameters a_1, a_2, a_3, although we used a limited set of data points, and a rather bad initial guess. However, it is always better to use large data sets of accurate measurements, and a good initial guess.

The reader should modify the initial guess in the above example, to see how both the results and the computation time are affected by a "bad" initial guess. For example, try the following cases:

1. `aguess={4.05,1.05,1.95}` is a very good initial guess; the algorithm converges rapidly to very accurate parameter values.

2. `aguess={2,0,1}` is a bad initial guess; the algorithm needs 11 iterations to converge, but the results are still very accurate.

3. `aguess={0,0,1}` is a very bad initial guess; the algorithm converges to a rather large value of χ^2 after 6 iterations, and the results obtained are clearly wrong.

```
require("LNAplot/PlotFunc","LNAplot/PlotData","LNA/LMfit")

local function f(x,a)
return math.sin(a[1]*x+a[2])+2*math.cos(a[3]*x)
end
local function dfda(x,a)
return {x*math.cos(a[1]*x+a[2]),math.cos(a[1]*x+a[2]),-2*x*math.sin(a[3]*x)}
end

local x,y,a,Da,chi2,axesdata

x={-1.5,-1,-0.5,0,0.5,1,1.7}
y={-1,-1,0.3,2.8,1.1,-1.8,-1}
chi2,a,Da=LMfit(f,dfda,{3,0,1},x,y)
print("\nFinal i|io: "..chi2)
print("\nParameters:")
for i,v in ipairs(a) do print(i,v) end
print("\nConfidences:")
for i,v in ipairs(Da) do print(i,v) end

axesdata=PlotData(x,y,{-2,2},{-2,3},false,1,3,0,"auto",true,1)
local function ffit(x)
return f(x,a) end
PlotFunc(ffit,{},axesdata,true)
```

Example program 15: XLMfit.

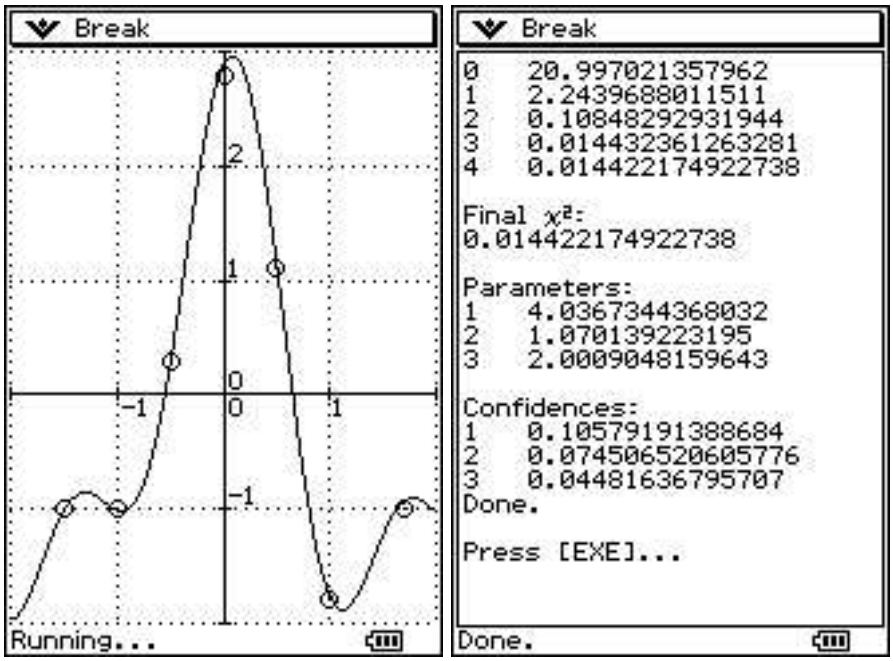


Figure 4.12: Results obtained by the example program XLMfit.

You should also try to modify the data points to see how the results are affected. For example, try to add an erroneous data point, or to use more accurate measurements.

It is worth emphasizing that the problem of non-linear fitting may have more than one correct solutions. For example, an initial guess equal to `aguess={3,0,-1}` (instead of `{3,0,1}`) will give the same results for the parameters a_1 and a_2, but a_3 is now computed as $a_3 = -2.00 \pm 0.04$; there is nothing wrong in this solution, since $\cos(a_3 x) = \cos(-a_3 x)$. In other words, if there are more than one solutions, `LMfit` does not have any way to "know" which solution is preferred; the solution that `LMfit` returns depends on the initial guess.

Remarks

Data stored in vector `xdata` need not to be sorted; unsorted x-data may be used, without affecting algorithm's behavior in any way. A good choice for the initial guess, `aguess`, is often crucial for the Levenberg-Marquardt algorithm. If the initial guess is far from the correct values, the algorithm may fail to converge, or it may converge to a wrong set of parameters. The most common error when using this function is to define `f` or `dfda` incorrectly; this will cause slow convergence to wrong results, so pay attention when defining these functions.

The "accuracy" `eps` refers to the minimized value for χ^2, not to the values of the parameters a; usually, there is no need to change its default value. The algorithm should converge after a few iterations (usually, less than 10); if it does not, it is almost sure that it will never converge, due to wrong initial guesses, or wrong input data. Therefore, you will probably never need to use the optional argument `maxit`.

FILENAME: LMfit.

DEPENDENCIES: `MatTrans`, `LUdecom`.

4.5 Numerical integration

4.5.1 TrapAdapt

The function `TrapAdapt` computes the definite integral of a function via the adaptive trapezoidal method. The adaptive trapezoidal method splits the integration interval into two subintervals and calls itself recursively, until the desired accuracy for the integral in each subinterval is satisfied.

Syntax

`q,err=TrapAdapt(f,a,b,`*`eps,warn ser,minstep`*`)`

returns the definite integral, `q`, of the function `f`, integrated from `a` to `b`; it also returns the estimated error, `err`. The arguments `eps`, `warnuser`, and `minstep` are optional: `eps` defines the desired accuracy (default: 10^{-6}); `warnuser` is a boolean argument that controls whether the user should be warned if the minimum step has been reached during calculations (default: `true`); `minstep` defines the minimum integration step (default: `Epsilon`, i.e., 1.12×10^{-16}).

Example

The example program XTrapAd uses the function `TrapAdapt` to compute the integral

$$\int_{-3}^{3} \left| \frac{x}{2} + \cos(2x) \right| .$$

In order to check the result, the absolute and relative errors are computed, given that the correct answer is 5.4909971377758389 (accurate to 16 decimal digits). The function to be integrated is shown graphically.

```
require("LNAplot/PlotFunc","LNA/TrapAd")

local function f(x) return math.abs(x/2+math.cos(2*x)) end

local q,qtru,err

qtru=5.4909971377758389
q=TrapAdapt(f,-3,3)
print("Computed integral:\n",q)
err=qtru-q
print("Absolute error:\n",err)
print("Relative error:\n",100*err/qtru.."%")
PlotFunc(f,{-3,3},{-0.5,2.5},true,1,"auto",true,1)
```

Example program 16: XTrapAd.

Figure 4.13 shows the results obtained by running this program. In this example, the integral is computed with default accuracy (10^{-6}), and the absolute error is indeed less than 10^{-6}.

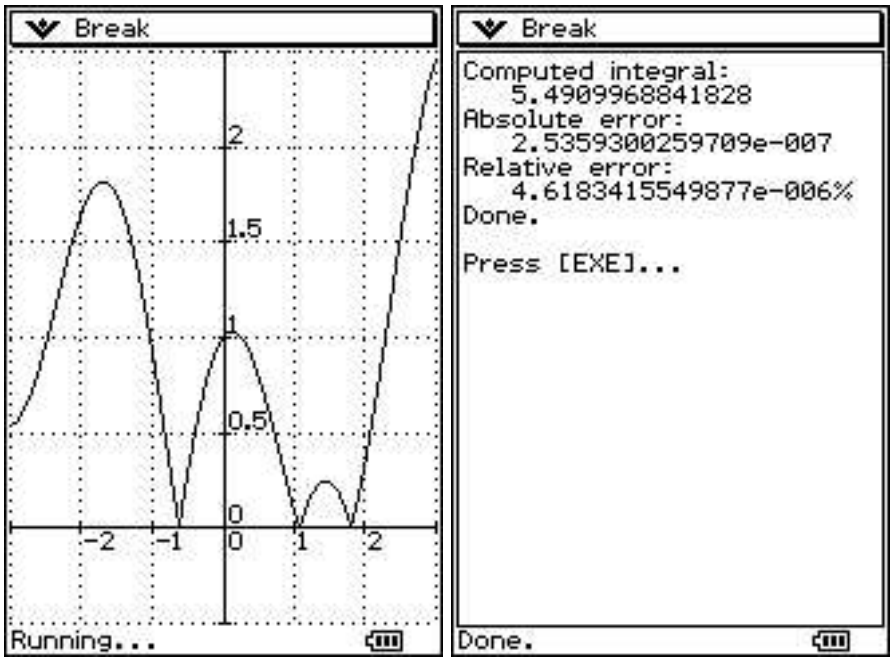


Figure 4.13: Results obtained by the example program XTrapAd.

Remarks

The estimated error is calculated per each integration subinterval. In some "pathological" cases, this error may be estimated as large, although the total error is very small (usually this happens in discontinuous functions, such as signal functions). In such cases, you may want to disable

the warning message by using `warnuser=false`. In most cases, the minimum integration step, `minstep`, does not need to be changed.

The function `TrapAdapt` should *not* be used if there are singularities within the integration interval.

FILENAME: TrapAd.

DEPENDENCIES: `Epsilon`.

4.5.2 Romberg

The function `Romberg` computes the definite integral of a function via the Romberg method. The Romberg method is one of the most powerful integration algorithms. It is often considered as the method of choice for numerical integration, provided that the function to be integrated is sufficiently smooth, and there are no singularities or discontinuities within the integration interval.

Syntax

`q=Romberg(f,a,b,`*`eps,k,show`*`)`

returns the definite integral, `q`, of the function `f`, integrated from `a` to `b`. The arguments `eps`, `k`, and `show` are optional: `eps` defines the desired accuracy (default: 10^{-6}); `k` defines the order of the method (default: 2); `show` is a boolean argument that controls whether an iteration progress will be displayed or not (default: `false`).

Example

The example program XRomberg uses the function `Romberg` to compute the integral

$$\frac{1}{\sqrt{2\pi}}\int_{-3}^{3} e^{-\frac{x^2}{2}}.$$

This is the integral of a normal distribution function. In order to check the result, the absolute and relative errors are computed, given that the correct answer is 0.9973002039367398 (accurate to 16 decimal digits). The function to be integrated is shown graphically.

```
require("LNAutils/Pi","LNAplot/PlotFunc","LNA/Romberg")

local function f(x) return math.exp(-x^2/2)/math.sqrt(2*Pi) end

local q,qtru,err

qtru=0.9973002039367398
q=Romberg(f,-3,3)
print("Computed integral:\n",q)
err=qtru-q
print("Absolute error:\n",err)
print("Relative error:\n",100*err/qtru.."%")
PlotFunc(f,{-3,3},{-0.1,0.5},true,1,"auto",true,1)
```

Example program 17: XRomberg.

Figure 4.14 shows the results obtained by running this program. Note that the absolute error of the result is about two orders of magnitude less than the desired accuracy (10^{-6}). This is not surprising; Romberg integration often gives very accurate results, eventhough the desired accuracy was not very high.

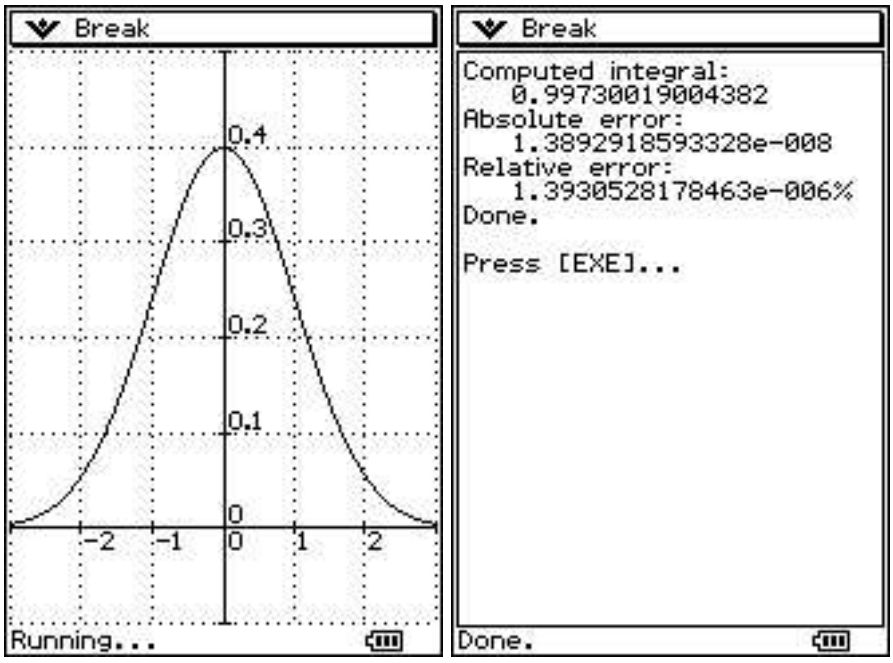


Figure 4.14: Results obtained by the example program XRomberg.

The reader should modify this example program, in order to compare the results obtained by `Romberg` with the corresponding results obtained by `TrapAdapt`.

Remarks

The order of the method, `k`, needs to be changed only rarely; the default value, `k=2`, means that the Simpson rule will be used iteratively to compute the integral. The function `Romberg` performs at most 20 iterations, and, if the accuracy `eps` is not reached, it returns the estimated integral, together with a warning message. There is no option to change the maximum number of iterations, because it is unlikely that more iterations will increase accuracy; furthermore, performing many iterations will probably lead to insufficient memory errors.

The Romberg method with error control is a very powerful integration method. In most cases, it gives very accurate results, and it is considerably faster than the adaptive trapezoidal method. Therefore, you should use `Romberg` instead of `TrapAdapt` to perform numerical integration. However, Romberg method is not suitable for integrating functions that are not sufficiently smooth, or for functions with discontinuities within the integration interval. In such cases, the function `TrapAdapt` might give more accurate results faster.

The function `Romberg` should *not* be used if there are singularities within the integration interval.

FILENAME: Romberg.

DEPENDENCIES: `Part`.

4.6 Ordinary differential equations

4.6.1 RK4Rich

The function `RK4Rich` solves numerically a first-order differential equation (or a system of first-order differential equations) via the Runge-Kutta method of fourth order with adaptive stepsize control via Richardson extrapolation. In other words, it solves an initial value problem, described by N differential equations

$$\begin{aligned} \frac{dy_1}{dx} &= f_1(x, y_1, y_2, \ldots, y_N), \\ \frac{dy_2}{dx} &= f_2(x, y_1, y_2, \ldots, y_N), \\ &\vdots \quad \vdots \quad \vdots \\ \frac{dy_N}{dx} &= f_N(x, y_1, y_2, \ldots, y_N), \end{aligned}$$

and a set of N initial conditions $y_1(x_i) = C_1, y_2(x_i) = C_2, \ldots, y_N(x_i) = C_N$, where x_i is the starting integration point, and $C_1, C_2, \ldots, C_N$ are constants. The solution is computed at a set of automatically selected $x-$points, within the user-specified interval $x \in [x_i, x_f]$.

`RK4Rich` is designed to solve first-order differential equations. However, higher-order differential equations can also be solved by using a simple transformation to a system of first-order differential equations. Similarly, systems of higher-order differential equations can be solved by transforming each differential equation to a set of first-order differential equations.

Runge-Kutta methods are widely used due to their efficiency. However, simple Runge-Kutta methods with fixed stepsize have an accuracy that depends on the integration stepsize. `RK4Rich` implements a more powerful Runge-Kutta method, where Richardson extrapolation is used to control the stepsize, so that the integration steps are selected automatically. This method is highly accurate, and has a very good error control. It is often considered as the method of choice for solving ordinary differential equations, provided that they are not extremely stiff.

Syntax

`yf,xp,yp=RK4Rich(RHS,xi,yi,xf,save,eps,maxit)`

integrates the differential equation(s) defined by the function `RHS` from $x =$ `xi` to $x =$ `xf`, with initial condition(s) `y=yi`. It returns the value of the function(s), `yf`, at the end point, and, optionally, a vector `xp`, containing the $x-$values selected by the algorithm, and a matrix `yp`, containing the corresponding function value(s); the first row of `yp` contains the values of the first function at the integration points `xp`, the second row contains the corresponding values for the second function, and so on. The function `RHS` should be written by the user, and defines the right-hand-side of the differential equation(s) to be solved. The argument `yi` is a vector defining the value of the functions at `x=xi`. The arguments `save` , `eps`, and `maxit` are optional: `save` is a boolean argument that controls whether the integration steps will be saved or not (default: `false`); if set to `true`, the vector `xp` and the matrix `yp` will be returned. `eps` defines the desired accuracy (default: 10^{-6}), and `maxit` defines the maximum number of integration points (default: 1000).

Example 1: Solving a system of first-order differential equations

Consider the initial value problem described by the system of differential equations

$$\frac{dy_1}{dx} = -3xy_1^2 + \frac{y_2}{1+x^3},$$

$$\frac{dy_2}{dx} = y_1 - \frac{xy_2}{5},$$

and the initial conditions $y_1(0) = 0.5$ and $y_2(0) = 1.5$. We want to find the solution, $y_1(x)$, $y_2(x)$, for $x \in [0,5]$. The example program XRK4Ric1 uses the function `RK4Rich` to solve the problem. The algorithm selects a number of appropriate integration points from $x = 0$ to $x = 5$. We set `savesteps=true`, so integration points are stored to the vector `xp`; the solution at these points is stored in the matrix `yp`.

```
require("LNAplot/PlotData","LNA/RK4Rich")

local function RHS(x,y)
return {-3*x*y[1]^2+y[2]/(1+x^3),y[1]-x*y[2]/5}
end

local y,xp,yp
y,xp,yp=RK4Rich(RHS,0,{0.5,1.5},5,true)
print(#xp.." integration points selected.")
print("Solution at end point:")
for i,v in ipairs(y) do print(i,v) end

PlotData(xp,yp,{0,5},{-0.1,2.5},true,0,0,{1,2},"auto",true,1)
```

Example program 18: XRK4Ric1.

Figure 4.15 shows the results obtained by running this program. In this example, the algorithm has selected 309 integration points (`#xp=309`, `xp[1]=0`, `xp[309]=5`). The values of $y_1(x)$ at these integration points are stored in the first row of the matrix `yp`, and the values of $y_2(x)$ are stored in the second row. The LNAplot function `PlotData` is used to show the results graphically. Thin line corresponds to $y_1(x)$, and thick line corresponds to $y_2(x)$.

Example 2: Solving a second-order differential equation

Consider the second-order differential equation

$$\frac{d^2y}{dx^2} = xy - y\frac{dy}{dx},$$

which must be solved with respect to the initial conditions $y(-1) = 0$ and $\frac{dy}{dx}\Big|_{x=-1} = 2$. Setting $y_1 = y$ and $y_2 = \frac{dy}{dx}$ transforms this equation to

$$\begin{aligned} \frac{dy_1}{dx} &= y_2, \\ \frac{dy_2}{dx} &= xy_1 - y_1y_2. \end{aligned}$$

The initial conditions are now written as $y_1(-1) = 0$, $y_2(-1) = 2$. Therefore, the problem has been reduced to a system of first-order differential equations. Solving this problem using `RK4Rich` is now straightforward. The example program XRK4Ric2 shows how the problem is solved. `RK4Rich` selects a number of appropriate integration points from $x = -1$ to $x = 3$.

Figure 4.16 shows the results obtained by running this program. In this example, the algorithm has selected 237 integration points (`#xp=237`, `xp[1]=-1`, `xp[237]=3`). The values of $y(x)$ at these

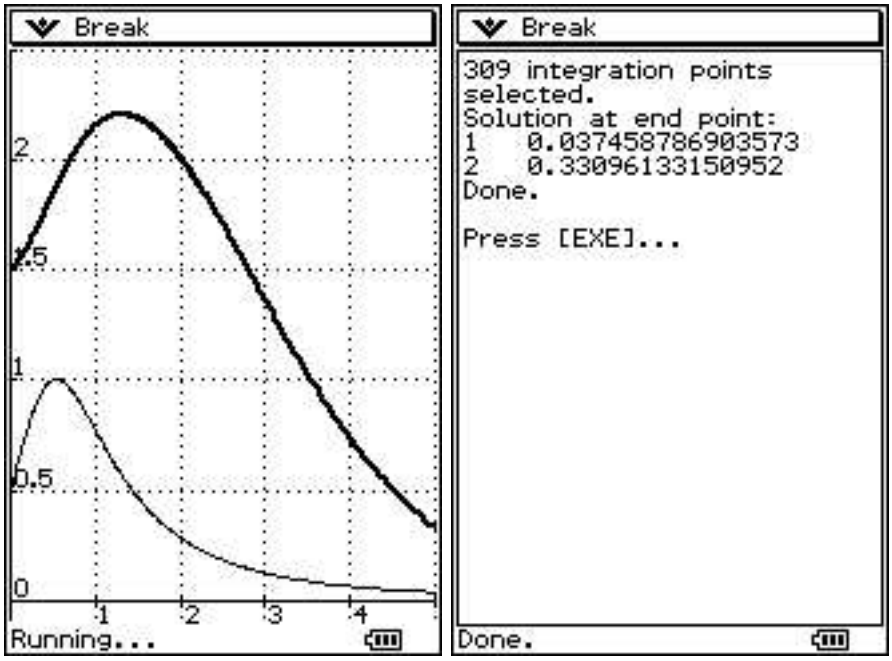


Figure 4.15: Results obtained by the example program XRK4Ric1.

```
require("LNAplot/PlotData","LNA/RK4Rich")

local function RHS(x,y)
return {y[2],x*y[1]-y[1]*y[2]}
end

local y,xp,yp
y,xp,yp=RK4Rich(RHS,-1,{0,2},3,true)
print(#xp.." integration points selected.")
print("Solution at end point:")
for i,v in ipairs(y) do print(i,v) end

PlotData(xp,yp,{-1,3},{-0.25,5},true,0,0,{1,2},"auto",true,1)
```

Example program 19: XRK4Ric2.

integration points are stored in the first row of the matrix `yp`, and the values of the derivative $\frac{dy}{dx}$ are stored in the second row. In the graphical representation of the results, thin line corresponds to $y(x)$, and thick line corresponds to $\frac{dy}{dx}$.

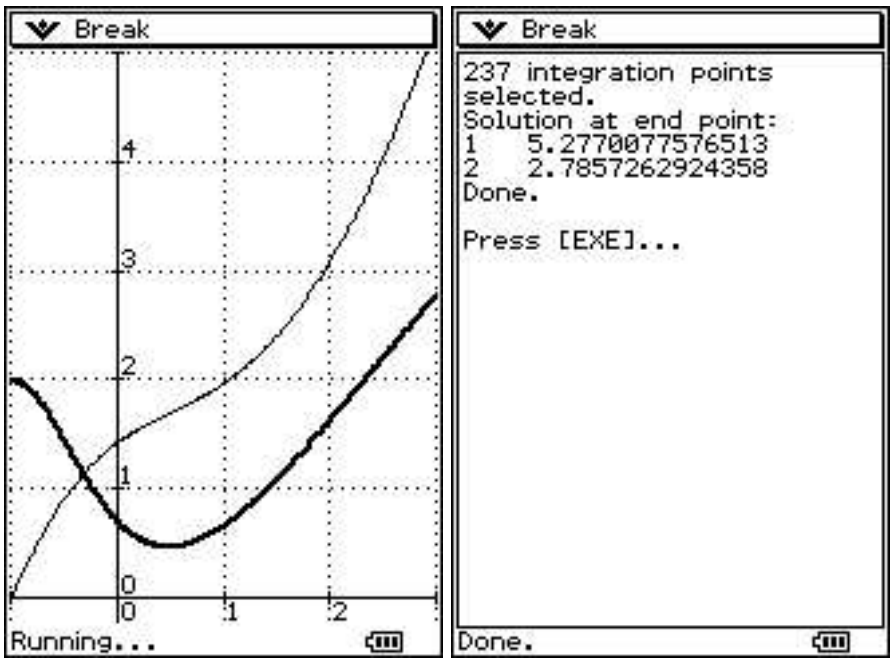


Figure 4.16: Results obtained by the example program XRK4Ric2.

Remarks

The starting integration point $x =$ `xi` needs not to be less than $x =$ `xf`. You can also reverse the direction of integration, by setting `xi` > `xf`. This is particularly useful when initial conditions are known at the right end point of the domain.

Computation time is typically a few seconds. In most cases, `RK4Rich` is powerful enough to return a result with an absolute error much less than the desired accuracy. Don't be surprised if you get a result with an absolute error of order $\sim 10^{-9}$ or less, although the preset accuracy, 10^{-6}, was used. However, the algorithm may need many integration points, especially if the integration interval is large. In this case, you may need to reduce accuracy, `eps`, or increase the maximum number of iterations, `maxit`. Note that, if `save` is set to `true`, you cannot set `maxit` to a very large number, due to memory constraints.

It is worth emphasizing that setting `save=true` is useful only if you need all intermediate results, i.e., the value(s) of the function(s) at each integration point. If you only need the solution at the end point, `x=xf`, you should not save intermediate steps. This way, a large amount of memory is saved, and you can solve problems that need a very large number of integration points.
FILENAME: RK4Rich.
DEPENDENCIES: None.

4.6.2 Shoot

The function `Shoot` solves a two-point boundary value problem, described by (a) a system of N first-order differential equations,

$$\frac{dy_1}{dx} = f_1(x, y_1, y_2, \ldots, y_N),$$

$$\begin{aligned} \frac{dy_2}{dx} &= f_2\left(x, y_1, y_2, \ldots, y_N\right), \\ \vdots \quad &\vdots \quad \vdots \\ \frac{dy_N}{dx} &= f_N\left(x, y_1, y_2, \ldots, y_N\right), \end{aligned}$$

(b) a set of N_i boundary conditions at $x = x_i$, and (c) a set of N_f boundary conditions at $x = x_f$, where x_i, x_f are the left and right end points of the integration interval, respectively. The total number of boundary conditions must be equal to the number of differential equations, i.e., $N_i + N_f = N$.

Alternatively, one or more differential equations of the second (or higher) order can be solved, provided that they can be converted to a system of first-order differential equations. `Shoot` is not restricted to simple Dirichlet or Neumann boundary conditions; more complicated Robin boundary conditions are also allowed.

This function is an implementation of the well-known *shooting method* for solving boundary value problems, hence its name. The shooting method is iterative, and it is widely used for its efficiency and speed. Used properly, `Shoot` is able to solve difficult boundary value problems in a rather short computation time, and with high accuracy. However, the reader should realize that solving boundary value problems is a *very* complicated task, and requires a large amount of computations. `Shoot` is a complicated function, and extensively uses "heavy" LNA functions, namely `RK4Rich` and `Broyden`. As a consequence, `Shoot` is currently the most memory-consuming function included in LNA.

The reader should be aware that, *used improperly, `Shoot` may fail to solve a simple problem, as easily as it solves difficult problems when used properly.* Furtheremore, `Shoot` can be called in many ways, depending on the problem at hand; this means that its syntax may confuse the novice user. Consequently, it is absolutely necessary to study all examples presented in this section before trying to use `Shoot` for the first time.

Syntax

`yimissing,yi,yf,xp,yp=Shoot(RHS,xi,yiguess,Init,xf,Bound,save,eps,maxit,show)`

finds a set of missing initial conditions, `yimissing`, so that the solution of the system of differential equations described by the function `RHS` satisfies a set of boundary conditions at $x =$ `xi` and $x =$ `xf`. As in `RK4Rich`, the user-provided function `RHS` should define the right-hand-side of the differential equations to be solved. `yiguess` is a vector, defining initial guesses for all missing initial conditions. `Init` is a user-defined function returning all initial conditions; it must be called as `Init(yiunknown)`, where `yiunknown` is a scalar, if there is only one missing initial condition, or a vector, if more than one initial conditions are missing. `Bound` is also a user-defined function, returning the boundary condition(s) at the end of the domain of integration; it must be called as `Bound(y)`, where `y` is the vector of the unknown functions. The function `Bound` should return a scalar, if there is only one known boundary condition at $x =$ `xf`, otherwise it should return a vector, containing all known boundary condition at $x =$ `xf`.

Some general examples will help understanding how the user-provided functions `Init` and `Bound` should be implemented. Assume that there are three differential equations, involving three unknown functions y_1, y_2, and y_3. Now consider the following cases:

1. There is one known boundary condition at $x =$ `xi` ($y_1 = 0$), and two known boundary conditions, at $x =$ `xf` ($y_2 = 5$, $y_3 = -7$). The function `Init` should return `{0, yiunknown[1], yiunknown[2]}`, where `0` means that y_1(`xi`) $= 0$, `yiunknown[1]` corresponds to the first missing initial condition, y_2(`xi`), and `yiunknown[2]` corresponds to second missing initial

condition, y_3(`xi`). The function `Bound` should return `{y[2]-5, y[3]+7}`, meaning that y_2(`xi`) $- 5 = 0$, and y_3(`xi`) $+ 7 = 0$. Note that all boundary conditions at $x =$ `xf` are written as $f(y_1, y_2, \ldots) = 0$.

2. There are two known boundary condition at $x =$ `xi` ($y_1 = 0$, $y_3 = -2$), and one known boundary condition, at $x =$ `xf` ($y_1 = 8$). The function `Init` should return `{0, yiunknown, -2}`, where `0` means that y_1(`xi`) $= 0$, `yiunknown` corresponds to the missing initial condition, y_2(`xi`), and `-2` means that y_3(`xi`) $= -2$. The function `Bound` should simply return `y[1]-8`, meaning that y_1(`xf`) $- 8 = 0$. Note that, in this case, `Bound` returns a scalar, since only one boundary condition is known at $x =$ `xf`.

3. There are two known boundary condition at $x =$ `xi` ($y_1 = 4$, $5y_2 + y_3 = 1$), and one known boundary condition, at $x =$ `xf` ($y_3 = 0$). In this case, the second boundary condition at $x =$ `xi` is a Robin boundary condition. In other words, we do not know y_2(`xi`) or y_3(`xi`) explicitly; instead, we know a relationship between them. We have two options: the unknown initial condition can be set to y_2(`xi`) or y_3(`xi`). In the first case, the function `Init` should return `{4, yiunknown, 1-5*yiunknown}`, where `4` means that y_1(`xi`) $= 4$, `yiunknown` corresponds to the missing initial condition, y_2(`xi`), and `1-5*yiunknown` means that y_3(`xi`) $= 1 - 5y_2$(`xi`). In the second case, the function `Init` should return `{4, (1-yiunknown)/5, yiunknown}`, where `yiunknown` corresponds to the missing initial condition, y_3(`xi`), and `(1-yiunknown)/5` means that y_2(`xi`) $= \frac{1-y_3(\texttt{xi})}{5}$. Whatever we choose as the missing initial condition, the function `Bound` should simply return `y[3]`, meaning that y_3(`xf`) $= 0$.

On output, `Shoot` returns the missing initial condition(s), `yimissing`, which is a scalar, if only there is only one missing initial condition, or a vector, if there are more than one missing initial conditions. `Shoot` also returns the vector `yi`, which is a complete set of appropriate initial conditions, as computed by the shooting method. As in `RK4Rich`, the optional boolean argument `save` controls whether the integration steps will be saved or not (default: `false`). If set to `true`, `Shoot` returns a vector `xp`, containing the $x-$values selected by the algorithm, and a matrix `yp`, containing the corresponding function value(s); the first row of `yp` contains the values of the first function at the integration points `xp`, the second row contains the corresponding values for the second function, and so on.

The arguments `eps`, `maxit`, and `show` are optional. `eps` is a vector with two elements; `eps[1]` is the required accuracy of the integration process, and `eps[2]` is the desired accuracy of the result. The default values for both accuracies is 10^{-6}. Similarly, `maxit` is a vector of two elements; `maxit[1]` is the maximum number of integration points (default: 1000), and `maxit[2]` is the maximum number of iterations (default: 20). Both `eps` and `maxit` can be set to `"auto"`, which is equivalent to the default values. `show` is a boolean argument that controls whether progress of the iterative process will be displayed or not (default: `true`).

Tips for selecting a good initial guess and improving performance

In general, a good choice for the initial guess, `yiguess`, is crucial for the shooting method. If the initial guess is bad, you may encounter problems; either the computation time may be large, or `Shoot` may not be able to return a solution, since the maximum number of integration points, `maxit[1]`, will be probably exceeded. In such cases, increasing `maxit[1]` is the last thing you should try. Instead, try to find a better initial guess. If you don't have any way to do that, try one or both of the following techniques.

1. Use a "trial" run of `Shoot`, with reduced accuracy for the integration process, `eps[1]`, and reduced accuracy for the result, `eps[2]`. This will give you a raw estimate of the missing initial conditions. Then run `Shoot` again, this time giving as initial guess the values obtained by the "trial" run. This technique is the first thing you should do whenever something goes wrong; there are many cases where it gives accurate results, while a single run of `Shoot` fails. Even if you are able to obtain a solution by simply calling `Shoot`, using a "trial" run of `Shoot` may reduce the computation time considerably. Compared to a simple call of `Shoot` with high accuracy, this technique will give equally accurate results, while the computation time is usually reduced to more than 50%.

2. If the problem has only one missing initial condition, try to "bracket" that condition, by using not just an initial guess, `{guess}`, but an interval `{{lguess,rguess}}`. In this case, `Shoot` will use the function `Brent`, instead of `NewtonR`, for finding the root. This will usually reduce the computation time. Furthermore, there are cases where providing a single initial guess may fail, while providing an interval for the initial guess gives accurate results. The main disadvantage of this technique is that there is no general way to find an interval `{{lguess,rguess}}`, containing the missing initial condition.

In some cases, a combination of the above techniques can reduce the computation time even further. If a particular problem has only one missing initial condition, you could use a "trial" run of `Shoot`, together with a bracketing interval for the missing initial condition. However, this combination usually affects the computation time only slightly.

Example 1: Solving a boundary value problem with two boundary conditions

Consider the boundary value problem described by the system of differential equations

$$\begin{aligned} \frac{dy_1}{dx} &= \sin y_1 + \cos y_2, \\ \frac{dy_2}{dx} &= \frac{y_2}{x + y_1^2}, \end{aligned}$$

and the boundary conditions $y_1(0) = 0.5$ and $y_1(3) = y_2(3)$. We want (a) to find the missing initial condition, $y_2(0)$, and (b) to find the solution, $y_1(x)$, $y_2(x)$, for $x \in [0, 3]$. All results should be accurate to at least five decimal digits. The example program `XShoot1a` uses the function `Shoot` to solve this problem, starting from the initial guess $y_2(0) = 1$. The results are shown graphically.

Note that, since our initial guess is $y_2(0) = 1$, the third argument of `Shoot` is `{1}`. Furthermore, since we require a result accurate to five decimal digits, both accuracies are set to 5×10^{-6}.

Figure 4.17 shows the results obtained by running this program. In this example, `Shoot` needs 6 iterations to find the missing initial condition. In the first iteration, the user-provided initial guess is used for the missing initial condition (in this case, $y_2(0) = 1$). In all other iterations, the algorithm selects an appropriate guess for $y_2(0)$. The "discrepancy" displayed is the value returned by `Bound` for each guess, i.e., it is equal to $y_1(3) - y_2(3)$. Notice how the discrepancy is getting smaller absolute values, as the algorithm approaches the missing initial condition. The iteration process is terminated when $y_2(0)$ is computed with sufficient accuracy (in this case, 5×10^{-6}). In the graphical representation of the results, thin line corresponds to $y_1(x)$, while thick line corresponds to $y_2(x)$. You can see that the missing initial condition is indeed computed accurately, since $y_1(3) \simeq y_2(3)$, and the discrepancy is $|y_1(3) - y_2(3)| \simeq 6.86 \times 10^{-12}$.
Note that, in this particular example, the value obtained for the missing initial condition is $y_2(0) = 0.83449967576416$. Thorough computations (using very accurate algorithms on a computer) show that this value is accurate to 8 decimal digits. In other words, we get a result which

```
require("LNAplot/PlotData","LNA/Shoot")

local function RHS(x,y)
return {math.sin(y[1])+math.cos(y[2]),y[2]/(x+y[1])^2}
end
local function Init(yiunknown) return {0.5,yiunknown} end
local function Bound(y) return y[1]-y[2] end

local yimissing,yi,yf,xp,yp

yimissing,yi,yf,xp,yp=Shoot(RHS,0,{1},Init,3,Bound,true,{5E-6,5E-6})

print("\nInitial conditions:")
for i,v in ipairs(yi) do print(i,v) end
print("Solution at end point:")
for i,v in ipairs(yf) do print(i,v) end
PlotData(xp,yp,{0,3},{0,2.5},true,0,0,{1,2})
```

Example program 20: XShoot1a.

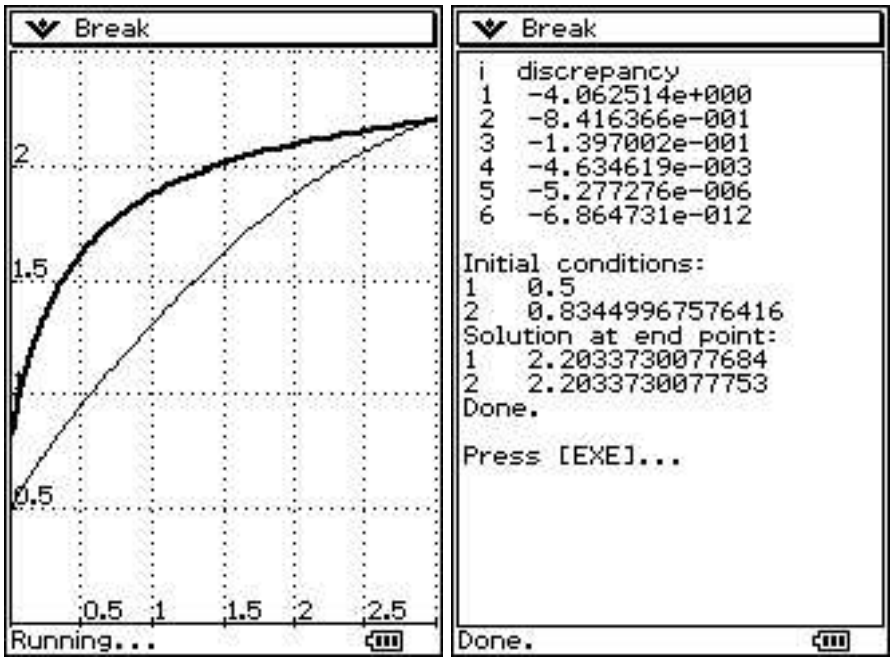


Figure 4.17: Results obtained by the example program XShoot1a.

is much more accurate than requested. This is not surprising; `Shoot` is often able to return a result with an absolute error much less than the desired accuracy.

In the example program XShoot1a, `Shoot` needs 1 minute and 29 seconds to compute the missing initial condition and return the solution. However, the computation time can be reduced considerably by using a "trial" run of `Shoot`, with reduced accuracies. The example program XShoot1b is a modification of XShoot1a, where a "trial" call of `Shoot` is used to find a raw estimation of the missing initial condition, $y_2(0)$. The raw estimation for the missing initial condition is stored in the variable `yilow`. Then the function `Shoot` is called again, with `{yilow}` as the initial guess. The reader should compare this example program with XShoot1a.

```
require("LNAplot/PlotData","LNA/Shoot")

local function RHS(x,y)
return {math.sin(y[1])+math.cos(y[2]),y[2]/(x+y[1])^2}
end
local function Init(yiunknown) return {0.5,yiunknown} end
local function Bound(y) return y[1]-y[2] end

local yilow,yimissing,yi,yf,xp,yp

-- Trial call of Shoot:
yilow=Shoot(RHS,0,{1},Init,3,Bound,false,{5E-3,5E-3})
print("\nRefined guess:",yilow)
waitkey();clear()
-- Final call of Shoot:
yimissing,yi,yf,xp,yp=Shoot(RHS,0,{yilow},Init,3,Bound,true,{5E-6,5E-6})

print("\nInitial conditions:")
for i,v in ipairs(yi) do print(i,v) end
print("Solution at end point:")
for i,v in ipairs(yf) do print(i,v) end
PlotData(xp,yp,{0,3},{0,2.5},true,0,0,{1,2})
```

Example program 21: XShoot1b.

Figure 4.18 shows the results obtained by running this program. The results obtained by the trial run are shown in figure 4.18a. Note that the trial run needs only 5 iterations, and it is terminated quickly, because both accuracies are reduced to 5×10^{-3}. We get the raw estimation $y_{2\,\mathrm{raw}}(0) = 0.83450010143322$, with an error less than 5×10^{-3}, as required. Figure 4.18b shows the results obtained by the second call of `Shoot`, where the initial guess is set to $y_{2\,\mathrm{raw}}(0)$. This is the most time-consuming part of the program, since both accuracies are set to 5×10^{-6}. However, only 2 iterations are needed, because the initial guess is good enough, so that the second run starts with low discrepancy, $|y_1(3) - y_2(3)| \simeq 5.00 \times 10^{-6}$.

Our final result is $y_2(0) = 0.83449967576410$, which is almost equal to the result obtained by the example program XShoot1a (and accurate to 8 decimal digits). However, the computation time for both the trial and the final call of `Shoot` is only 42 seconds. In other words, using a trial call of `Shoot`, we were able to obtain a very accurate result, while the computation time has been reduced to about 53%.

In this example, there is only one missing initial condition. Therefore, we could also "bracket" the missing initial condition, by providing an interval where the missing initial condition lies. To

(a) Trial call of `Shoot`. (b) Final call of `Shoot`.

Figure 4.18: Results obtained by the example program XShoot1b.

see how bracketing works, try the following modifications:

1. In the example program XShoot1a, replace the initial guess, `{1}`, with `{{0,1}}`. This means that $0 \leq y_2(0) \leq 1$. Having this additional information, `Shoot` will use the function `Brent` (instead of `NewtonR`) to find the missing initial condition. With this modification, the computation time is reduced to 52 seconds, well less than the computation time needed by the previous version of example program XShoot1a.

2. In the example program XShoot1b, replace the initial guess, `{1}` with `{{0,1}}`. This brackets the raw estimation of the missing initial condition, $0 \leq y_{2\,\mathrm{raw}}(0) \leq 1$. Furthermore, we can also bracket the initial guess in the second call of `Shoot`. Since the raw estimation $y_{2\,\mathrm{raw}}(0)$ has an error less than 5×10^{-3}, we can use a narrow interval as the initial guess in the second call. To do that, replace the initial guess `{yilow}` with `{{yilow-5E-3,yilow+5E-3}}`. This means that $y_{2\,\mathrm{raw}}(0) - 5 \times 10^{-3} \leq y_2(0) \leq y_{2\,\mathrm{raw}}(0) + 5 \times 10^{-3}$. With this modification, the computation time is 41 seconds, which is almost equal to the computation time needed by the previous version of example program XShoot1b. In other words, a combination of both special techniques affects the computation time very slightly.

Example 2: Solving a boundary value problem with three boundary conditions

Consider the boundary value problem described by the system of differential equations

$$\begin{aligned} \frac{dy_1}{dx} &= y_1 - y_2 + y_3, \\ \frac{dy_2}{dx} &= \frac{x^2 y_1 + y_2}{x + y_3^2}, \end{aligned}$$

$$\frac{dy_3}{dx} = -y_1^2 y_3 - x y_2,$$

and the boundary conditions $y_3(0) = -2$ and $y_1(2.5) = 3$, $y_3(2.5) = 4y_2(2.5)$. We want (a) to find the missing initial conditions, $y_1(0)$, $y_2(0)$, and (b) to find the solution, $y_1(x)$, $y_2(x)$, $y_3(x)$ for $x \in [0, 2.5]$. As in Example 1, all results should be accurate to at least five decimal digits. The example program XShoot2a uses the function `Shoot` to solve this problem, starting from the initial guess $y_1(0) = 0$, $y_2(0) = 0$. The results are shown graphically.

```
require("LNAplot/PlotData","LNA/Shoot")

local function RHS(x,y)
return {y[1]-y[2]+y[3],(x^2*y[1]+y[2])/(x+y[3]^2),-y[1]^2*y[3]-x*y[2]}
end
local function Init(yiunknown) return {yiunknown[1],yiunknown[2],-2} end
local function Bound(y) return {y[1]-3,y[3]-4*y[2]} end

local yimissing,yi,yf,xp,yp

yimissing,yi,yf,xp,yp=Shoot(RHS,0,{0,0},Init,2.5,Bound,true,{5E-6,5E-6})

print("\nInitial conditions:")
for i,v in ipairs(yi) do print(i,v) end
print("Solution at end point:")
for i,v in ipairs(yf) do print(i,v) end
PlotData(xp,yp,{0,2.5},{-2,3},true,0,0,{1,2,3},"auto",true,1)
```

Example program 22: XShoot2a.

Figure 4.19 shows the results obtained by running this program. In this example, `Shoot` needs 12 iterations to find the missing initial condition. In the first iteration, the user-provided initial guess is used for the missing initial conditions (in this case, $y_1(0) = 0$, $y_2(0)$). In all other iterations, the algorithm selects an appropriate guess for $y_1(0)$ and $y_2(0)$. The "max discrepancy" displayed is the maximum absolute value of the discrepancies vector, i.e., it is equal to $\max(|y_1(2.5) - 3|, |y_3(2.5) - 4y_2(2.5)|)$. The iteration process is terminated when both missing initial conditions are computed with desired accuracy (in this case, 5×10^{-6}). In the graphical representation of the results, thin line corresponds to $y_1(x)$, thicker line corresponds to $y_2(x)$, and thickest line corresponds to $y_3(x)$. The missing initial conditions are indeed computed accurately, since $y_1(2.5) \simeq 3$, and $y_3(2.5) \simeq 4y_2(2.5)$.
The values obtained for the missing initial conditions are $y_1(0) = 0.46946638871335$, and $y_2(0) = -0.87898247727826$. Thorough computations show that these valuse are accurate to 6 decimal digits. As in the previous example, the results are more accurate than required.

In this example, `Shoot` needs 4 minutes and 22 seconds to compute the missing initial conditions. This large computation time is caused by several reasons: there are two missing initial conditions, the initial guess for $y_2(0)$ is rather bad, and we have set the accuracy to 5×10^{-6}, which is rather excessive for this particular problem. However, the computation time can be reduced considerably by using a "trial" run of `Shoot`. The example program XShoot2b is a modification of XShoot2a, where a "trial" call of `Shoot` is used to find a raw estimation of the missing initial conditions, $y_1(0)$ and $y_2(0)$. This estimation is used as the initial guess in a second call of `Shoot`.

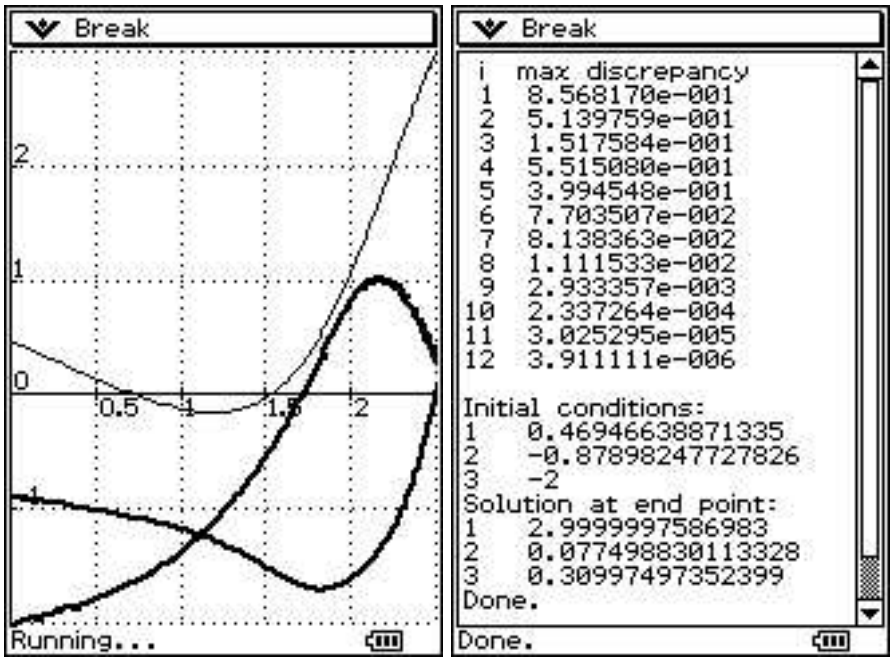
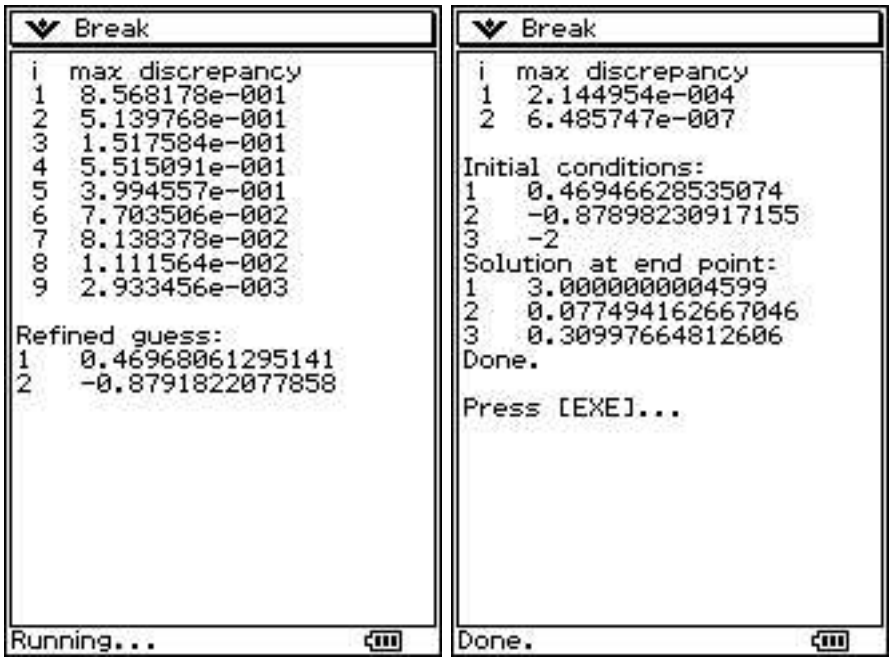


Figure 4.19: Results obtained by the example program XShoot2a.

```
require("LNAplot/PlotData","LNA/Shoot")

local function RHS(x,y)
return {y[1]-y[2]+y[3],(x^2*y[1]+y[2])/(x+y[3]^2),-y[1]^2*y[3]-x*y[2]}
end
local function Init(yiunknown) return {yiunknown[1],yiunknown[2],-2} end
local function Bound(y) return {y[1]-3,y[3]-4*y[2]} end

local yilow,yimissing,yi,yf,xp,yp

-- Trial call of Shoot:
yilow=Shoot(RHS,0,{0,0},Init,2.5,Bound,false,{5E-3,5E-3})
print("\nRefined guess:")
for i,v in ipairs(yilow) do print(i,v) end
waitkey();clear()
-- Final call of Shoot:
yimissing,yi,yf,xp,yp=Shoot(RHS,0,yilow,Init,2.5,Bound,true,{5E-6,5E-6})

print("\nInitial conditions:")
for i,v in ipairs(yi) do print(i,v) end
print("Solution at end point:")
for i,v in ipairs(yf) do print(i,v) end
PlotData(xp,yp,{0,2.5},{-2,3},true,0,0,{1,2,3},"auto",true,1)
```

Example program 23: XShoot2b.

Figure 4.20a shows the results obtained by the trial call of `Shoot`, where the both accuracies are reduced to 5×10^{-3}. The trial run is terminated after 9 iterations, yielding the raw estimations $y_{1\,\mathrm{raw}}(0) = 0.46968061295141$ and $y_{2\,\mathrm{raw}}(0) = -0.8791822077858$, which are stored in the vector `yilow`. Figure 4.20b shows the results obtained by the second call of `Shoot`, where the initial guess is set to `yilow`, and both accuracies are set to 5×10^{-6}. Only 2 iterations are needed, since the initial guess is good enough, so that the second run starts with low maximum discrepancy, $\max(|y_1(2.5) - 3|, |y_3(2.5) - 4y_2(2.5)|) \simeq 2.14 \times 10^{-4}$.



(a) Trial call of `Shoot`. (b) Final call of `Shoot`.

Figure 4.20: Results obtained by the example program XShoot2b.

The final values for the missing initial conditions are $y_1(0) = 0.46946628535074$ and $y_2(0) = -0.87898230917155$. These values are almost equal to those obtained by the example program XShoot2a. In fact, they are even more accurate, since thorough computations show that missing initial conditions are accurate to 9 decimal digits. In other words, using a trial run of `Shoot` does not reduce accuracy; on the contrary, the accuracy of the results may be improved (as in this example). The computation time for both the trial and the final call of `Shoot` is 1 minute, 44 seconds, which means that, compared with the example program XShoot2a, the computation time has been reduced to about 60%.

Remarks

`Shoot` is powerful enough to handle Robin boundary conditions, such as $ay_1(\mathtt{xi}) + by_2(\mathtt{xi}) = c$. In other words, boundary value problems where not even one function is known explicitly at $x = \mathtt{xi}$ can be solved. However, it is well known that Robin conditions at $x = \mathtt{xi}$ are more "sensitive" to the initial guess. This means that solving a boundary value problem may be a considerably more difficult task, if there are Robin condition(s) at the starting integration point, and the initial guess is bad. `Shoot` is usually able to solve such problems without a good initial guess, but the computation time may be large. If there are one or more Robin initial conditions at $x = \mathtt{xi}$, while

there are no Robin conditions at $x =$ `xf`, you may consider to reverse the integration direction. In some cases, reversing the integration direction may give accurate results, even if the initial guess is bad.

It is worth emphasizing that, if you are unable to solve a given problem, you should check the definitions of the functions `Init` and `Bound` before trying anything else; even a small error in these functions leads to a problem which is different than what you wanted to solve. Often, this new problem cannot be solved by `Shoot`, simply because there is no solution.

FILENAME: Shoot.

DEPENDENCIES: RK4Rich, Broyden, Brent, NewtonR.

5 LNA change log

5.1 Version 1.00 (September 19, 2005)

- Initial version, including four numerical methods (`Bisect`, `Brent`, `KroneRoots`, and `Romberg`). All functions should be used in CPLua version 0.61 or above.

- The utility library contains seven utility functions (`EpsilonC`, `MaxLoc`, `MinLoc`, `Part`, `Printf`, `Sign`, and `Nint`), and one utility constant (`Epsilon`).

5.2 Version 1.10 (September 23, 2005)

- A method for solving ordinary differential equation(s) has been added (`RK4Rich`).

- The organization of the library has been changed; all numerical methods, together with their driver programs, are now included in a single directory named LuaNumAn.

- The documentation has been reorganized and slightly changed.

5.3 Version 1.20 (September 28, 2005)

- The package now includes a plotting library, called LuaPlot. By making use of this library, graphical representation of the results has been added to most driver programs. The LuaPlot library is still in development, and it is more than likely that the appearance of the graphics produced by this library will be changed in the future.

- Minor modifications in almost all numerical methods have been made, in accordance with the new features of CPLua version 0.71. All these modifications are not apparent to the user. This version is not compatible with previous CPLua versions.

- A utility function has been added (`Column`).

- The function `Romberg` has been modified. It is now slightly faster and more accurate.

- The function `KroneRoots` has been modified. It is now remarkably faster than previous versions (about 50% faster than version 1.00).

- The function `RK4Rich` has been slightly modified, concerning its output. It now returns a matrix `yp`, where each row (not column, as before) contains the values of each function at the integration points `xp`. This is more convenient in practice.

- The utility function `Sign` now becomes obsolete, and has been removed. CPLua version 0.71 provides the built-in function `math.sign` which has exactly the same functionality.

5.4 Version 1.30 (October 5, 2005)

- Minor modifications have been made, in accordance with the new features of CPLua version 0.72. All these modifications are not apparent to the user. This version is not compatible with previous CPLua versions.
- Four utility functions, concerning matrix operations, have been added (`MatIdent`, `MatMul`, `MatPrint`, and `MatTrans`). The utility function `Column` has been renamed to `MatCol`, so that all matrix-related utility functions have the prefix "`Mat`".
- A method for LU decomposition of a matrix and its applications (solving a system of linear equations, computing the determinant or the inverse of a matrix) has been added (`LUdecompose`, `LUsubstitute`, `LUsolve`, `LUdeterminant`, and `LUinverse`).
- A method for non-linear fitting of data has been added (`LMfit`).
- Plotting functions now can use labels.

5.5 Version 1.40 (October 25, 2005)

- A method for numerical integration has been added (`TrapAdapt`).
- LU decomposition has been slightly modified, and it now handles singular matrices more conveniently.
- The function `Romberg` has been slightly changed. It now returns the estimated integral, even if the desired accuracy has not been reached (in this case, a warning message is also printed).
- A menu-driven demo program (`DemoAll`) has been added.

5.6 Version 1.50 (March 15, 2006)

Version 1.50 has many new features and improvements:

- Several functions have been rewritten to a more compact code, and modifications in almost all numerical methods have been made, in accordance with the new features of CPLua version 0.8. All these modifications are not apparent to the user. This version is not compatible with previous CPLua versions.
- The organization of the library has been changed. The directory LuaNumAn now contains only functions implementing numerical methods, so that there is no need to be accessed by the user. All driver programs have been moved to the directory LuaDrive. In addition, simple examples, demonstrating how to use each numerical method, have been added to the directory LuaExamp.
- Two utility functions (`OrderMag`, `LinSpace`) and a utility constant (`Pi`) have been added.
- The plotting functions `PlotFunc` and `PlotData` have been improved. They now use a much better automatic selection of tics spacing so that, usually, there is no need to set the tics spacing manually. Furthermore, `PlotFunc` can now handle user-defined discontinuities, declared by the new optional argument `discont`.

- A plotting function for "infinite" plots has been added (`PlotInf`).
- A method for solving tridiagonal systems of linear equations has been added (`Tridiag`).
- A method for performing cubic spline interpolation has been added (`CSpline`).
- A method for numerically computing the Jacobian of a multivariate function has been added (`Jacobian`).
- A method for solving systems of non-linear equations has been added (`Broyden`).
- The documentation has been reorganized and considerably improved. Example programs have been added for all functions, together with useful remarks and guidelines.

5.7 Version 1.60 (June 12, 2006)

The name of the project has been changed to LNA. Note that the directories LuaNumAn, LuaUtils, LuaPlot, LuaExamp, and LuaDrive have been renamed to LNA, LNAutils, LNAplot, LNAexamp, and LNAdrive, respectively. In version 1.60, all numerical methods initially planned for the project have been implemented. This version includes many new features and improvements:

- Minor modifications and improvements have been made in almost all programs of the package, including the example and driver programs. Most of these modifications are not apparent to the user. This version is not compatible with previous CPLua versions.
- The utility function `Printf` now becomes obsolete, and has been removed. CPLua provides the built-in function `printf` which has exactly the same functionality.
- Two utility timing functions have been added (`TimeDiff` and `TimeElapsed`).
- The optional argument `warnuser` has been added to the function `CubicSpline`, so that warning messages in case of extrapolation can be switched off.
- The optional argument `show` has been added to the functions `Bisect`, `Brent`, and `LMfit`, so that the user can select whether intermediate computations will be displayed or not.
- The function `Broyden` has been slightly modified: if the optional argument `show` is set to true, the maximum discrepancy at each iteration is displayed, including the first (in the previous version, the maximum discrepancy was displayed starting from the second iteration).
- An implementation of the Newton-Raphson method for computing the root of a function has been added (`NewtonR`).
- A method for performing linear interpolation has been added (`LinInterp`).
- A method for solving two-point boundary value problems has been added (`Shoot`).
- The directory LNAtest has been added in the package. It includes a "test" program, useful for testing future versions of LNA or CPLua.

6 Bibliography

The following list presents some books that have been widely used during the implementation of the LNA package.

1. Brian Bradie, *A Friendly Introduction to Numerical Analysis*, Pearson Prentice Hall, New Jersey 2006.

   An excellent introductory book, with plenty of thoroughly selected exercises and applications in Physics, Chemistry, Statistics, and other sciences. The book also includes thorough theoretical discussions.

2. Jean-Pierre Nougier, *Méthodes de calcul numérique* vols. 1 and 2, Hermes, Paris 2001.

   This book includes many numerical methods with full theoretical discussion. A pseudocode and an example is given for each numerical method. Differences between numerical methods are also discussed in detail.

3. Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri, *Méthodes numériques pour le calcul scientifique*, Springer-Verlag, Paris 2000.

   This book includes a deep theoretical discussion for each numerical method presented.

4. William H. Press, Saul A. Teukolsky, William T. Vellering, and Brain P. Flannery, *Numerical Recipes - The Art of Scientific Computing*, Second Edition, Cambridge University Press, Cambridge 1992.

   A best-seller in Numerical Analysis books. It covers plenty of numerical methods, and includes a rather short theoretical discussion. An implementation of each numerical method is given in Fortran 90, Fortran 77, or C (depending on the version of the book).

5. T. M. R. Ellis and Ivor R. Philips, *Programming in F*, Addison-Wesley Longman, Essex 1998.

   An excellent book for learning module-oriented programming in Fortran 95. All LNA functions are written according to the programming style suggested in this book. Although this is not a book about Numerical Analysis, it contains several introductory numerical methods, explained in detail.

7 The GNU General Public License

The following text is the GNU General Public License (GPL), as published by the Free Software Foundation. If you are planning to modify and/or distribute the source code of LuaNumAn, you should follow its terms.

GNU GENERAL PUBLIC LICENSE Version 2, June 1991.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain

entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING

RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS