LuaNumAn version 1.10

September 23, 2005

Contents

1 Introduction

1.1 What is LuaNumAn?

LuaNumAn is a library of Numerical Analysis functions, together with several utility functions. All functions included in the library are meant to be called from a Lua program written in CPLua version 0.61 or above, which runs on a Casio ClassPad 300. This document explains the functions currently included in LuaNumAn. This project is continuously developed and it is more than likely that new Numerical Analysis functions and/or library functions will be added in the future. Existing functions may also be modified in a future version, but compatibility with previous version will be respected, unless it is absolutely necessary to modify the way a function is called.

1.2 Disclaimer and Copyright

LuaNumAn is free software. You can redistribute it and/or modify it under the terms of the GNU General Public License, as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. The functions included in the software have been thoroughly tested and debugged, and the author will be surprised if there are still bugs. However, the author is not fool to claim that this software is 100% bug-free, and there is probably no developer who wants to claim such a thing for his/her programs. This software is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose. See the complete GNU General Public License for more details (section 5 of this document, page 8).

LuaNumAn, together with this document, is Copyright © 2005 by PAP. The CPLua Add-In for ClassPad 300 is written and maintained by Orwell.

1.3 Library organization

This software is consisting of functions organized in two directories: The LuaUtils directory, containing utility functions, and the LuaNumAn directory, containing the Numerical Analysis functions currently included in the software. For each numerical method, there are two files in the directory LuaNumAn:

1. A file containing all functions needed to implement a specific numerical method. 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 auxiliary functions that are needed in the computations, but these are hidden by 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.

2. An example Lua program (a "driver" program), showing how this function can be used. This file is named with the letter "D" and the name of the function, or an abbreviation of this name, if its length is more than seven characters.

For example, the main function for the Romberg method is named `Romberg`. This function, together with all auxiliary functions needed for this method, is defined in the file Romberg, which is located at the directory LuaNumAn. The driver program for the Romberg method is implemented in the file DRomberg, also located in the directory LuaNumAn.

Almost all Numerical Analysis functions in LuaNumAn use other functions (numerical methods and/or utility functions), i.e., they have dependencies. For example, a 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 LuaNumAn function, you may also need several other functions included in the library. Therefore, *you should not delete or modify any file included in the LuaUtils or in the LuaNumAn directory*, unless you know what you are doing.

2 Utility functions

The LuaUtils directory includes several utility functions written in Lua. Most of them are called in the Numerical Analysis functions included in this software.

2.1 Utility constants

2.1.1 Epsilon

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

2.2 Utility functions

2.2.1 EpsilonC

DESCRIPTION: The function `EpsilonC` computes `Epsilon`, the smallest positive number ϵ, for which $1 + \epsilon > 1$.
SYNTAX: `EpsilonC(show,eps,maxit)` returns `Epsilon`. 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 LuaUtils constant `Epsilon` is more than sufficient.

2.2.2 MaxLoc

DESCRIPTION: 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: If you just want to compute the maximum value (not its position), you can use `math.max(unpack(A))` instead.

2.2.3 MinLoc

DESCRIPTION: 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: If you just want to compute the minimum value (not its position), you can use `math.min(unpack(A))` instead.

2.2.4 Part

DESCRIPTION: 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`, `Part` returns an empty vector, `{}`.

2.2.5 Printf

DESCRIPTION: The function `Printf` simulates the C function `printf` to print formatted text and/or variables.
SYNTAX: `Printf(form,var1,var2,...)` prints the arguments `var1`, `var2`,... using the format string `form`.
EXAMPLE: `Printf("pi accurate to %i digits:  %.3f",4,3.14157)` prints:
`pi accurate to 4 digits:  3.142`.
REMARKS: The format string `form` may include text to be printed, as well as format specifiers and escape characters. Most common format specifiers are `%i`, `%f`, `%e`, and `%s`, used to print integers, float numbers in decimal form, float numbers in exponential form, and strings, respectively. The most common escape character is `\n` which corresponds to a line feed. See any C or Lua manual for more details on format strings.

2.2.6 Sign

DESCRIPTION: The function `Sign` returns a number `a` with the same sign as number `b`.
SYNTAX: `Sign(a,b)` returns `|a|` if `b>0`, `-|a|`, if `b<0`, and `0`, if `b==0`.
EXAMPLES: `Sign(3.14,-2)` returns `-3.14`; `Sign(-3.14,2)` returns `3.14`.
REMARKS: This a generalized version of the function `sign`, available in several languages. Most often, it is used with `1` as its first argument, returning `-1`, `0`, or `1`, depending on the second argument.

2.2.7 Nint

DESCRIPTION: The function `Nint` returns the nearest integer to its argument.
SYNTAX: `Nint(x)` returns the nearest integer to `x`.
EXAMPLES: `Nint(1.86)` returns 2; `Nint(-3.2)` returns `-3`.

3 Numerical Analysis functions

This section describes all Numerical Analysis functions currently included in LuaNumAn. 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 accompanying driver program before using any function is strongly recommended.

You should realize that all functions included LuaNumAn 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 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.

All functions described in this section 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 preset default value. When describing the syntax of a particular function, optional arguments are written in italic characters.

Almost all the functions included in LuaNumAn 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. 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, and 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).

3.1 Root finding

3.1.1 Bisect

DESCRIPTION: The function `Bisect` computes the root of a function within a given interval via the Bisection method.
DETAILS: 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`*`)` returns the `root` of the function `f` inside the interval `{xl,xr}`, together with an estimation of the absolute `error`. The arguments `eps` and `maxit` are optional: `eps` defines the desired accuracy (default: `Epsilon`, i.e., 1.12×10^{-16}), and `maxit` defines the maximum number of iterations (default: 100).

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). 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`, `Sign`.

3.1.2 Brent

Description: The function `Brent` computes the root of a function within a given interval via the Brent method.

Details: 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 a similar syntax as `Bisect`.
Syntax: `root=Brent(f,xl,xr,`*`eps,maxit`*`)` returns the `root` of the function `f` inside the interval `{xl,xr}`. The arguments `eps` and `maxit` are optional: `eps` defines the desired accuracy (default: `Epsilon`, i.e., 1.12×10^{-16}), and `maxit` defines the maximum number of iterations (default: 100).
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`, `Sign`.

3.1.3 KroneRoots

Description: The function `KroneRoots` computes all the roots of a function within a given interval, or, optionally, a predescribed number of roots.
Details: The function `KroneRoots` 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 LuaNumAn. 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 predescribed 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`.

REMARKS: Setting the desired number of roots, `r_req`, may be useful if you only need some of the roots, 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, but, in some cases, 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`.

3.2 Numerical integration

3.2.1 Romberg

DESCRIPTION: The function `Romberg` computes the definite integral of a function via the Romberg method.
DETAILS: The Romberg method is one of the most powerful integration algorithms. It is often considered as the method of choice for numerical integration.
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: 1×10^{-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`).
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` should *not* be used if there are singularities within the integration interval.
FILENAME: Romberg.
DEPENDENCIES: `Part`.

3.3 Solution of ordinary differential equation(s)

3.3.1 RK4Rich

DESCRIPTION: The function `RK4Rich` solves 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.
DETAILS: The 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 number of steps. The function `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: `y,xp,yp=RK4Rich(RHS,xi,yi,xf,`*`save_steps,eps,maxit`*`)` integrates the differential equation(s) defined by the function `RHS` from the point `x=xi` to the point `x=xf`, with initial condition(s) `y=yi`. It returns the value of the function(s), `y`, at the end point, and, optionally, a vector `xp`, containing the values of `x` taken, and a matrix `yp`, containing the corresponding

function value(s); the first column of `yp` contains the values of the first function at the integration points `xp`, the second column 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_steps` , `eps`, and `maxit` are optional: `save_steps` is a boolean argument that controls whether the integration steps will be saved or not (default: `false`; if enabled, the vector `xp` and the matrix `yp` will be returned); `eps` defines the desired accuracy (default: 1×10^{-6}); `maxit` defines maximum number of iterations (default: 1000).
REMARKS: In most cases, this function is powerful enough to return a result with an absolute error much less than the desired accuracy; don't be surprised if you use the preset accuracy, 10^{-6}, and you get a result with an absolute error of the order $\sim 10^{-9}$ or less. Setting `savesteps=true` is useful only if you need the intermediate results (not just the value of function(s) at `x=xf`). The default maximum number of iterations is usually more than enough to get a highly accurate result, so the argument `maxit` is rarely used. The computation time is typically a few seconds.
FILENAME: RK4Rich.
DEPENDENCIES: None.

4 LuaNumAn change log

4.1 Version 1.00 (September 19, 2005)

- Initial version of LuaNumAn, including four numerical methods (`Bisect`, `Brent`, `KroneRoots`, and `Romberg`).
- The LuaUtils library contains seven utility functions (`EpsilonC`, `MaxLoc`, `MinLoc`, `Part`, `Printf`, `Sign`, and `Nint`), and one utility constant (`Epsilon`).

4.2 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 The GNU General Public License

GNU GENERAL PUBLIC LICENSE Version 2, June 1991.

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

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.)

NO WARRANTY

END OF TERMS AND CONDITIONS