

Solis

Environment for Numerical Computing

Sidi HAMADY

Full Professor, Dr. habil. Eng.

Université de Lorraine, France

LMOPS Lab., Université de Lorraine & CentraleSupélec, France

<http://www.hamady.org>

The latest Solis release (extracted size less than 10 MB) is freely available here:

http://www.hamady.org/download/solis_windows_64bit.zip

http://www.hamady.org/download/solis_linux_64bit.tgz

The latest version of this manual:

http://www.hamady.org/download/solis_lua.pdf

Solis:

Copyright(C) 2010-2023 Prof. Sidi HAMADY

<http://www.hamady.org>

sidi@hamady.org

Solis is protected by copyright laws and international copyright treaties, as well as other intellectual property laws and treaties. Solis is free of charge only for non-commercial use. Sidi Ould Saad Hamady expressly disclaims any warranty for Solis. Solis is provided 'As Is' without any express or implied warranty of any kind, including but not limited to any warranties of merchantability, noninfringement, or fitness of a particular purpose. Solis may not be redistributed without authorization of the author.

Solis Lua engine uses:

the Lua programming language, (C) Lua.org, PUC-Rio.

the Lua Just-In-Time Compiler, (C) Mike Pall.

the LIS linear solvers, (C) The SSI Project, Kyushu University, Japan.

the ODE solver developed by Scott D. Cohen and Alan C. Hindmarsh @ LLNL.

Solis device editor, code editor, data plotter and calculator use:

the IUP GUI toolkit, (C) Tecgraf/PUC-Rio.

the Scintilla Component, (C) Neil Hodgson.

Contents

Solis	i
Contents	ii
Listings	iii
Presentation	1
Solis Editor	2
Solis Modules	4
Time functions	4
Math functions	5
Numerical Array Functions	5
Ordinary Differential Equations	8
Mathematical Optimization	8
Descriptive Statistics	10
Data Analysis	11
ASCII Data Files	13
XY Data Plotting	14
BSD Socket	17
C Modules	20
Instrumentation	23
Math Parser	23
Graphical User Interface using IUP	30
Solis Calculator	32
Bibliography	35

Listings

1	build.bat File to Compile L ^A T _E X Document under Windows.	4
2	build.sh File to Compile L ^A T _E X Document under Linux.	4
3	Time Functions.	5
4	Using Time Functions.	5
5	Using Math Functions.	5
6	Lua Math Functions.	6
7	Extended Math Functions.	6
8	Constants (universal constants in international units (SI)).	6
9	Ordinary Differential Equations (ODE).	9
10	Mathematical Optimization.	10
11	Descriptive Statistics.	11
12	The Lua Function used by <code>data.fit</code>	11
13	Data Fitting Example.	12
14	FFT Example.	13
15	Loadind and Saving ASCII Data.	14
16	XY Data plotting.	16
17	Using BSD Sockets.	20
18	Pattern Matching Module.	21
19	AES Encryption Module.	21
20	Library for Arbitrary Precision Math (MAPM).	22
21	Library for XML.	22
22	C Module Template.	24
23	Generate Lib File from <code>libluacore.dll</code> for Visual C++.	24
24	Module Makefile.	25
25	Loading a C Module.	25
26	VISA Example.	25
27	RS232 Example.	26
28	Using the Math Parser.	30
29	Graphical User Interface using IUP.	31
30	Mathematical Functions in SolisCalc.	33
31	Constants in SolisCalc.	33
32	Commands in SolisCalc.	34

Presentation

Solis is a programming environment for numerical computing and data analysis using the Lua scripting language [1, 2]. It is available for Linux and Windows with built-in Lua scripting engine, integrated numerical, data plotting and analysis modules, full-featured editor with syntax highlighting, code completion, online documentation, code samples, etc.

The main goal of Solis is to provide an easy-to-use development environment for numerical computing using the Lua programming language on Linux and Windows. It integrates the Lua scripting engine with all Lua functionalities, and a lot of specific Solis functionalities including numerical functions (differential equations, mathematical optimization, etc.), data plotting and analysis modules and an extended mathematical library. In addition, it includes modules for instrumentation using VISA and serial interfaces. Within the Solis environment, you can develop algorithms for science and engineering with one of the most elegant and fast scripting languages. To learn the Lua programming language, you can read the reference book[1] and visit the Lua official website:

<http://www.lua.org>

Excellent tutorials, covering all Lua aspect from basics to advanced programming techniques, can be found here:

<http://lua-users.org/wiki/TutorialDirectory>

The Solis editor can also be used as a general purpose full-featured editor supporting C/C++, Bash/Text, Python, Octave, Fortran, LaTeX and Makefile with configurable tools (e.g. to run a compiler or a bash script).

To install Solis for Windows or Linux, download `solis_windows_64bit.zip` (Windows 64bit) or `solis_linux_64bit.tgz` (Linux 64bit) from <http://www.hamady.org>, unzip/untar in any location (USB key or Memory stick for example) and run `solisedit` (Linux) or `solisedit.exe` (Windows) in the bin directory.

The Solis distribution includes:

- An advanced code editor, **solisedit** under Linux or **solisedit.exe** under Windows, implemented in C. This editor offers all functionality found in modern editors such as syntax highlighting, autocompletion, markers, indentation control, find/replace, file explorer... and are fully customizable. It supports Lua, Python, C/C++, L^AT_EX... and can be used as a general code editor. It can also be used to edit and run semiconductor device simulations using Solis Simulator.
- A data plotter, **solisplot** (Linux) or **solisplot.exe** (Windows), implemented in C and C++. This tool is used by Solis but could also be used as a standalone data plotter.
- An advanced scientific calculator, **soliscalc** under Linux or **soliscalc.exe** under Windows, implemented in C.
- A semiconductor device simulator [3], driven by **soliscomp** under Linux or **soliscomp.exe** under Windows, controlling the semiconductor simulator implementing the drift-diffusion

model. The complete simulator documentation is located in [solis_simulator.pdf](#). I started developing the simulator in 2009 and presented the first testing release in the 37th International Symposium on Compound Semiconductors (ISCS) in 2010 in Japan [4].

- A graphical device editor, **solisdevice** under Linux or **solisdevice.exe** under Windows, implemented in C. This tool gives an easy-to-use graphical frontend to the semiconductor device simulator.

The whole Solis distribution size, including documentation and examples, is less than 10 MB. To know if a new version is available, click *Menu/Help/Check for Update...* or visit my website: <http://www.hamady.org>

Under Linux, Solis includes also an interactive terminal emulator (**solisterm**), a standalone version of the embedded terminal in SolisEdit.

This terminal emulator is loaded and available to use if the VTE library is installed.

This Solis user manual is organized as follows:

The first chapter contains the Solis editor.

The second chapter gives an overview on how to use the Solis environment.

The third chapter contains the description of the Solis built-in modules.

The fourth chapter contains the description of the scientific calculator.

Solis Editor

The Solis code editor, **SolisEdit**, offers all functionality found in modern editors such as syntax highlighting, autocompletion, markers, indentation control, find/replace, file explorer... and is fully customizable (screenshot in figure 1).

SolisEdit can be used to edit the Solis input files (with extension **.solis**) and model files (with extension **.lua**) and, in addition, it supports a set of languages used by scientists and engineers such as C/C++, Bash, Python, Octave, Fortran, \LaTeX ¹ and Makefile. The language is automatically selected based on the file extension.

SolisEdit integrates a **File Explorer** to work easily with files, that can be set to show only (filter) some files based on their extension (right-click the root directory and select the corresponding filter).

SolisEdit includes a system of coloured **Markers**, shown in the margins, to show in realtime the modified/saved sections of the current documents. The **Markers** can be reset at any time by

¹This manual was composed in \LaTeX using **SolisEdit**.

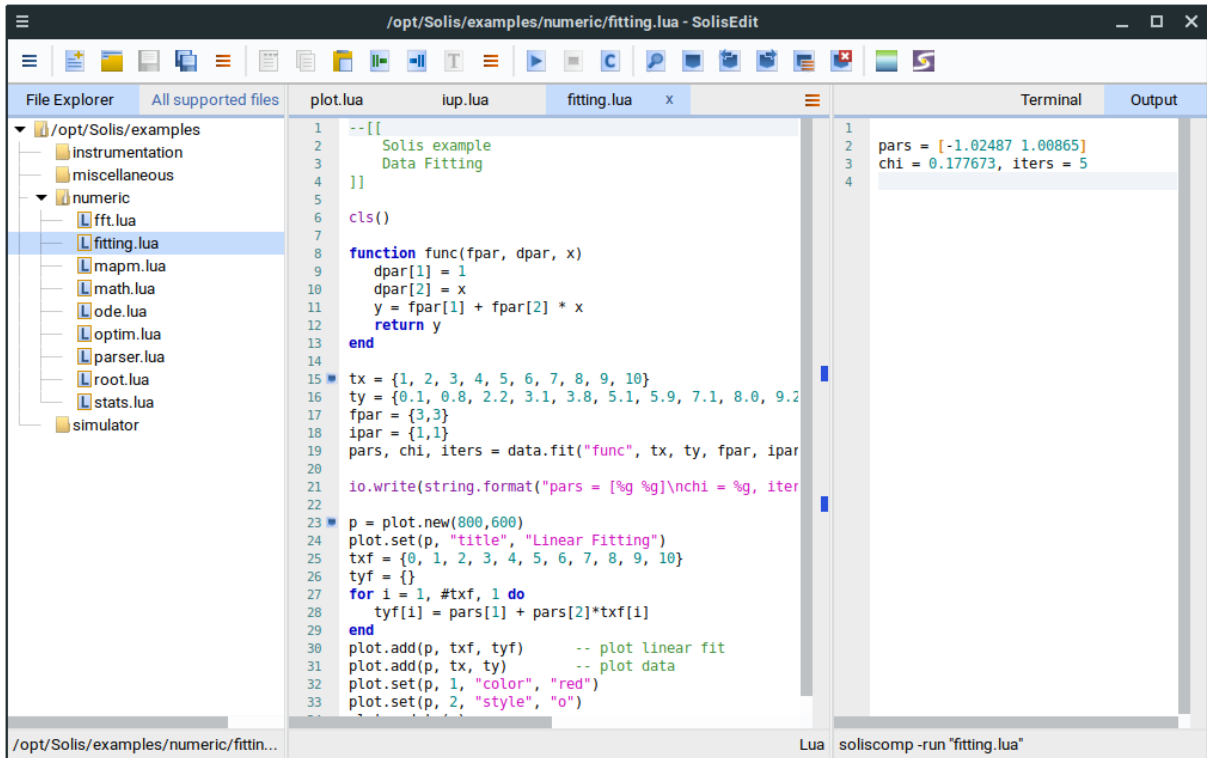


Figure 1: SolisEdit Screenshot.

selecting *Edit/Remove* Markers menu.

One particular marker is the bookmark, used to mark a specific line in the code for easier navigation.

The classic Search/Replace functionality are included in SolisEdit: *Search* menu.

Almost every aspect of the SolisEdit user interface can be customized: *Options* menu.

With SolisEdit, one can configure specific tool, such as a compiler or a Bash script, to run for a known file type (Solis and Lua of course, but also C, Python, \LaTeX , etc.). To configure a tool, select *Tools/Compiler* menu and type the command to use to build/compile the corresponding file type. For example, for Solis, the command is `solis -run %s.solis` where `%s` will be replaced by the filename. It is useful to call a batch file (under Windows) or bash script (under Linux) to build a specific file. This can be done in the following way:

- For Windows, one can create a batch file (named `build.bat` for example), put the command in this file and add the batch name `build.bat` in the *Tools/Compiler* dialog after selecting the corresponding file type (example: \LaTeX). The example in **Listing 1** gives a typical `build.bat` content for building \LaTeX document under Windows using MiKTeX and the Sumatra PDF viewer.
- For Linux, similarly you can create a Bash file (named `build.sh` for example), put the command in this file and add `./build.sh` in the *Tools/Compiler* dialog after selecting the

```
@echo off
@del /f /q myreport.pdf >nul 2>&1
latexmk -pdf -pvc- -halt-on-error myreport.tex
if %errorlevel% equ 0 (
    start "" "SumatraPDF.exe" myreport.pdf
) else (exit /b 1)
```

Listing 1: build.bat File to Compile L^AT_EX Document under Windows.

```
rm -f myreport.pdf >/dev/null 2>&1
latexmk -pdf -pvc- -halt-on-error myreport.tex
RETSTATUS=$?
if [ $RETSTATUS -eq 0 ]; then
    xdg-open myreport.pdf >/dev/null 2>&1 &
fi
```

Listing 2: build.sh File to Compile L^AT_EX Document under Linux.

corresponding file type (example: L^AT_EX) as shown in **Listing 2**.

- For Python (for both Windows or Linux), just put a command like this (replace with your installed Python interpreter):

```
python -u %s.py
```

In the *Tools/Compiler* dialog, one can check the *Redirect standard output* option to let SolisEdit to print out all the text generated by the tool to the Output Window. If this option is unchecked, SolisEdit will launch a command window and run the tool inside it. In this latter case, one can check the *Close when restart* option to close the previous command window before starting a new one. For the Solis simulator it is better to check the *Redirect standard output* option (it is checked, by default) to benefit from functionality such as syntax coloring.

Under Linux, SolisEdit includes an embedded terminal emulator. This terminal emulator is loaded and available to use if the VTE library is installed.

Solis Modules

The Solis environment integrates the Lua scripting engine with all Lua functionalities, and a lot of specific Solis functionalities presented below.

Time functions

An example on how to use the time functions is given in **Listing 4**.


```
time.tic()           -- sets the wall-clock timer
time.toc()          -- returns the number of milliseconds elapsed
                    -- since the last tic() call
time.sleep(n)       -- sleeps for n milliseconds
time.format(n)      -- formats a duration (integer in milliseconds)
                    -- into string
```

Listing 3: Time Functions.

```
-- Time
time.tic()
time.sleep(200)
dt = time.toc()
print(time.format(dt))
```

Listing 4: Using Time Functions.

Math functions

With Solis, the math functions are mapped to global functions (e.g.: you can use `cos` or `math.cos`). An example is shown in **Listing 5**.

Listing 6 give a summary of the Lua math functions and the Solis global math functions.

NB: Lua gives the Napierian logarithm the name `log` and the decimal logarithm is named `log10`, as in C language.

Special math functions are added to Solis (`math` namespace), including and extending the Lua math functions.

A summary of math additional functions and constants is given in **Listing 7** and **Listing 8**

Numerical Array Functions

Functions to handle numerical arrays that can be called through the `linalg` namespace. Below is a summary of the **linalg** functions:

```
-- Math
cls()
a = cos(pi/4)
print("result = ", a)
```

Listing 5: Using Math Functions.

```

math.abs          math.acos          math.asin
math.atan         math.atan2         math.ceil
math.cos          math.cosh         math.deg
math.exp          math.floor        math.fmod
math.frexp        math.huge         math.ldexp
math.log          math.log10        math.max
math.min          math.modf         math.pi
math.pow          math.rad          math.random
math.randomseed   math.sin          math.sinh
math.sqrt         math.tanh         math.tan

```

Listing 6: Lua Math Functions.

```

math.exp2(x)      -- 2^x
math.logb(x)      -- log base 2 of x
math.cbrt(x)      -- cubic root
math.hypot(x,y)   -- sqrt(x^2 + y^2)
math.erf(x)       -- error function
math.erfc(x)      -- complementary error function
math.lgamma(x)    -- ln(gamma(x))
math.tgamma(x)    -- gamma(x)
math.trunc(x)     -- nearest integer
math.round(x)     -- nearest integer, rounding
math.isinf(x)     -- number is infinite?
math.isnan(x)     -- not a number?
math.isnormal(x)  -- number is normal?
math.asinh(x)     --
math.acosh(x)     --
math.atanh(x)     --
math.gauss(x,b,c) -- G(x) = exp(-(x - b)^2 / 2c^2)
math.lorentz(x,b,c) -- L(x) = (c / ((x - b)^2 + c^2))

```

Listing 7: Extended Math Functions.

```

math.q            -- Electron charge (in C)
math.me           -- Electron mass (kg)
math.kb           -- Boltzmann constant (J/K)
math.h            -- Planck constant (Js)
math.c            -- Speed of Light in vacuum (m/s)
math.na           -- Avogadro constant (1/mole)

```

Listing 8: Constants (universal constants in international units (SI)).

x = linalg.array(n,a)

create array of size n and initialize its values to a

x = linalg.zeros(n)

create n-size array and initialize its values to zero

x = linalg.ones(n)

create n-size array and initialize its values to one

linalg.swap(x,y)

swap two arrays

linalg.copy(x,y)

copy array x to y

s = linalg.dot(x,y)

scalar product

z = linalg.add(x,y,a,b)

return $a*x + b*y$

y = linalg.get(x,is,ie)

return x subarray from index is to ie

z = linalg.cat(x,y)

concatenate x and y arrays

B = linalg.transpose(A,m)

transpose a matrix with m lines

s = linalg.format(A,m)

format an array with m lines and return a string

x = linalg.rand(n,rmin,rmax)

return an array of random values between rmin and rmax

i = linalg.imin(x)

return index of the min value in x

i = linalg.imax(x)

return index of the max value in x

s = linalg.sum(x)

return the array sum

d = linalg.norm(x)

return the array norm-2

Ordinary Differential Equations

With the Solis ODE solver can integrate differential system given the system functions, the initial values and the independent variable value (x, t, ...). It is not necessary to provide the Jacobian which is approximated by the solver.

y = ode.solve(func, y0, t0, t1, tol)

Integrates ODE system, where:

func: name of the ODE system function. func defined as func(t, y, ydot) where:

ydot vector updated with respect to y and t.

y0 table containing the initial values

t0 value for y0

t1 value to integrate for

tol the solver tolerance (optional)

An example is given in **Listing 9**.

Mathematical Optimization

Solis integrates a mathematical optimization module including minimization of real function of n variables. The Solis minimization function uses the Hooke and Jeeves algorithm which do not require the Jacobian to be evaluated.

iters = optim.minimize(func, pars, maxiters, tol, rho)

func: name of the function of n variables to be minimized.

func defined as func(x) where:

```
-- Damped Oscillator:  $y'' + c y' + k y = 0$ 
local c = 0.5
local k = 1
function func(t, y, ydot)
    ydot[1] = y[2]
    ydot[2] = (-c * y[2]) + (-k * y[1])
end

-- Solve with 0.1 seconds as interval
y0 = {2, 0}
t0 = 0
dt = 0.1
t1 = t0 + dt
tol = 1e-3
tm = {}
y = {}
dy = {}
for i = 1,100,1 do
    yy = ode.solve("func", y0, t0, t1, tol)
    tm[i] = t1
    y[i] = yy[1]
    dy[i] = yy[2]
    t0 = t1
    t1 = t1 + dt
    y0[1] = yy[1]
    y0[2] = yy[2]
end

-- plot solution y and y'
p = plot.new(800, 600)
plot.add(p, tm, y)
plot.add(p, tm, dy)
plot.set(p, "xlabel", "time (s)")
plot.set(p, "ylabel", "amplitude")
plot.set(p, 1, "legend", "y(t)")
plot.set(p, 1, "color", "red")
plot.set(p, 2, "legend", "y'(t)")
plot.set(p, 2, "color", "blue")
plot.update(p)
```

Listing 9: Ordinary Differential Equations (ODE).

```

-- Optimization
function booth(x)
    return (math.pow(x[1] + 2*x[2] - 4, 2) + math.pow(2*x[1] + x[2] - 5, 2))
end

x = { -5, 5 }
iters = optim.minimize("booth", x, 100, 1e-6)

cls()
io.write(string.format("\n x = [%g %g]  iters = %d \n", x[1], x[2], iters))

function func(x)
    return x^2 - x - 1
end
x = optim.root("func", -5, 5)    -- expected: x ~ -0.618034
print(x, func(x))

```

Listing 10: Mathematical Optimization.

x vector containing the variables

pars table containing the initial values (will be updated to the calculated values)

maxiters maximum number of iterations (optional)

tol the algorithm tolerance (optional)

rho the algorithm parameter, between 0 and 1 (optional). Decrease rho to improve speed and increase it for better convergence.

xr = optim.root(func,a,b,iters,tol)

– Finds the root of function in the [a,b] interval (optional), the given maximum number of iterations (optional) and tolerance (optional).

Returns the root value.

An example is given in **Listing 10**.

Descriptive Statistics

The **data** namespace includes functions to calculate the descriptive parameters of a list of values:

Minimum: **data.min(t)**

Maximum: **data.max(t)**

Sum: **data.sum(t)**

Mean: **data.mean(t)**

Median: **data.median(t)**

```
-- Stats
cls()
t = {1,1,2,3,4,4,5}
m = data.mean(t)      -- expected: m = 2.8571428571429
print(m)
```

Listing 11: Descriptive Statistics.

```
function fitfun(fpar, dpar, x)
  dpar[1] = 1      -- fpar is the fitting parameters table
  dpar[2] = x      -- dpar is the table of partial derivatives.
  y = fpar[1] + fpar[2]*x  -- x is the independent variable
  return y
end
```

Listing 12: The Lua Function used by `data.fit`.

Variance: `data.var(t)`

Standard Deviation: `data.dev(t)`

Coefficient of Variation: `data.coeff(t)`

Root Mean Square: `data.rms(t)`

Skewness: `data.skew(t)`

Kurtosis excess: `data.kurt(t)`

All the stats functions take a Lua table as argument, with 2048 maximum number of elements.

An example is given in **Listing 11**.

Data Analysis

The `data` namespace includes functions to perform data analysis including fitting using user-defined model, FFT and autocorrelation calculations:

`pars, chi, iters, str = data.fit(func, tx, ty, fpar, ipar, tol, iters)`

runs the fitter algorithm with:

func the Lua model function name. The Lua function syntax is shown in **Listing 12** (replace with your own model).

tx is the table with X data

ty the table with Y data.

fpar is the fitting parameters table.

ipar is the table containing, for each parameter, value 1 if the parameter is varying or 0 if it is fixed. this parameter ipar is optional.

```

-- Linear Fit
cls()
function func(fpar, dpar, x)
    dpar[1] = 1
    dpar[2] = x
    y = fpar[1] + fpar[2] * x
    return y
end

tx = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
ty = {0.1, 0.8, 2.2, 3.1, 3.8, 5.1, 5.9, 7.1, 8.0, 9.2}
fpar = {3,3}
ipar = {1,1}
pars, chi, iters = data.fit("func",tx,ty,fpar,ipar,1e-3,100)
io.write(string.format("pars = [%g %g]\n", pars[1], pars[2]))

p = plot.new(800,600)
plot.set(p, "title", "Linear Fitting")
txf = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
tyf = {}
for i = 1, #txf, 1 do
    tyf[i] = pars[1] + pars[2]*txf[i]
end
plot.add(p, txf, tyf)          -- plot linear fit
plot.add(p, tx, ty)          -- plot data
plot.set(p, 1, "color", "red")
plot.set(p, 2, "style", "o")
plot.update(p)

```

Listing 13: Data Fitting Example.

tol is the relative tolerance to be reached. this parameter **tol** is optional.

iters is the maximum number of iterations for the fitting algorithm. this parameter **iters** is optional.

The function `data.fit` returns four parameters: the obtained parameters table **pars** ; the chi number ; the number of performed iterations **iters** and a message **str** from the fitter engine.

An example of using the fitter is given in **Listing 13**.

ft = data.fft(data, idir)

calculates the FFT with:

data is the table with data

idir 1 for forward FFT and 0 for inverse

The function `data.fft` returns the obtained FFT table **ft**.

NB: the FFT amplitude is scaled (divided) by the number of points.

An example of using `data.fft` is given in **Listing 13**.


```

-- FFT
cls()

Fo = 50           -- signal frequency (Hz)
To = 1/Fo        -- signal period (seconds)
A = 5            -- signal amplitude
An = 1           -- noise amplitude
N = 256          -- number of points (power of 2)
Ts = 4 * To/N    -- sampling period
Fs = 1/Ts        -- sampling frequency
f = {}
t = {}
y = {}
for i = 1, N, 1 do
    f[i] = (i - 1) * Fs / (N - 1)      -- frequency
    t[i] = (i - 1) * Ts                -- time
    y[i] = A*cos(2*pi*Fo*t[i]) + An*lmath.random()
end

tfd = data.fft(y, 1)
p = plot.new(800,600)
plot.add(p, f, tfd)
plot.update(p)

```

Listing 14: FFT Example.

ac = data.acorr(data)

calculates the autocorrelation with data is the table with data.

The function data.acorr returns the obtained autocorrelation table ac.

yf = data.filter(x, y, forder)

Filters (smooths) x-y data using the Savitzky-Golay method, given the filter order. Returns the filtered data yf.

ASCII Data Files

c1,c2,... = data.load(filename, sep, skip, colcount, rowcount)

Loads ASCII data with:

filename source file name.

sep separator , usually tab or semicolon (optional).

skip number of rows to be skipped (optional).

colcount number of columns to load (optional).

rowcount number of rows to load (optional).

The function data.load returns tables containing numeric data c1, c2, ...

rowcount = data.save(filename, sep, header, c1, c2, ...)

```
-- ASCII

cls()

fname = "C:\\Temp\\ascii.txt"
x = {1, 2, 3, 4, 5}
y = {1, 2, 3, 4, 5}
sep = "\t"
skip = 0
rc = data.save(fname, sep, "HEADER\n", x, y)
print(rc, "\n")

xt, yt = data.load(fname, sep, skip)
print(xt, "\n")
print(yt)
```

Listing 15: Loading and Saving ASCII Data.

Saves numeric data to ASCII file with:

filename: destination file name

sep: separator (usually tab or semicolon).

header: file header (comment, labels, ...)

c1, c2, ...: tables to save

The function `data.save` returns the number of rows actually saved rowcount.

An example of using `data.load` and `data.save` is given in **Listing 15**.

XY Data Plotting

The plot namespace includes functions to plot data:

p = plot.new(width, height, template)

creates plot.

plot.add(p, x, y)

adds a curve with tables x,y.

plot.add(p, x, y, ey)

adds a curve with tables x,y and error bar table ey.

plot.add(p, x, y, ex, ey)

adds a curve with tables x,y and error bar tables ex,ey.

plot.add(p, curve, x, y)

adds tables x,y to an existing curve.

plot.add(p, curve, x, y, ey)

adds tables x,y to an existing curve, with error bar table ey.

plot.add(p, curve, x, y, ex, ey)

adds tables x,y to an existing curve, with error bar tables ex,ey.

plot.add(p, expr, npoints, xstart, xend)

adds a curve with expression (like "sin(x)", "1 - exp(-x)" ...).

plot.add(p, fname)

adds a curve from data file (columns 1 and 2, TAB separated).

plot.rem(p, curve)

removes curve from plot.

plot.set(p, curve, prop, val)

sets the curve properties. prop can be:

"size" for curve line and symbol size. val is the size ("1", "2")

"style" for curve. val is "o" for circle, "+" for plus sign, "s" for square, "d" for diamond, and "-" for line. Example: "-s" for line and squared markers.

"legend" for curve legend. val is the legend text.

"color" for curve color. val is color name like "red", "blue", ... or hex-value like "FF0000").

plot.set(p, prop, val)

sets the plot properties. prop can be:

"title" for the plot window title. val is the window title.

"xlabel" for the bottom axis label. val is the axis label.

```

-- Plot
x = {0,1,2,3,4,5}
y = {0,1,2,3,4,5}
p = plot.new(800, 600)           --create a plot
plot.set(p, "title", "Plot example")
plot.add(p, x, y)               --add curve to the new plot
plot.set(p, 1, "size", 2)       --set curve #1 line size
plot.set(p, 1, "style", "-o")   --set curve #1 style(line and marker)
plot.set(p, 1, "color", "0000FF") --set curve #1 color
plot.update(p)

```

Listing 16: XY Data plotting.

"ylabel" for the left axis label. val is the axis label.

"xscale" for the bottom axis scale: val is "log" or "linear".

"yscale" for the left axis scale: val is "log" or "linear".

"xlim" for the bottom axis scale: val is "[min,max]" where min and max are the x-axis limits.

"ylim" for the left axis scale: val is "[min,max]" where min and max are the y-axis limits.

"maxpoints" set val as the maximum number of points per curve.

"autolim". val is "true": automatically set the axis limits.

plot.save(p, fname)

saves plot to file (SVG or PDF).

plot.update(p)

updates the plot window.

plot.close(p)

closes the plot window.

An example of XY data plotting is given in **Listing 16**.

All the plot properties (curves options, scale, axis, colors ...) can be easily modified in the plot window. All these properties can be saved in a style template file to be used later. One can create a style template for a particular plot type and use it in the Lua code (plot.new function). You can also add text, lines, rectangles, ellipses ... to the plot and save it as PDF or SVG directly from within the plot window.

BSD Socket

The socket namespace includes BSD-like network functions:

s = socket.new(af,type,proto)

creates socket where:

af: family (socket.AF_INET by default)

type: type (socket.SOCK_STREAM by default)

proto: protocol (def: socket.IPPROTO_TCP)

returns the socket identifier

ok = socket.bind(s, addr, port)

binds socket s, where:

s: socket identifier

addr: address (ex: socket.INADDR_ANY)

port: port to bind to

returns true on success

ok = socket.listen(s, backlog)

listens on socket, where:

s: socket identifier

backlog: maximum queue length

returns true on success

ok = socket.connect(s, addr, port)

connects socket, where:

s: socket identifier

addr: address

port: port to connect to

returns true on success

sa = socket.accept(s, addr, port)

accepts connection and create new socket:

s: socket identifier

returns the new socket identifier `sa`

ok = socket.timeout(s, to)

sets the recv and send timeout, where:

s: socket identifier

timeout in milliseconds

returns true on success

ok = socket.setsockopt(s, opt, val)

sets socket option, where:

s: socket identifier

option to set (ex: socket.SO_SNDTIMEO)

val: option value

returns true on success

ip = socket.getpeername(s)

gets the socket s peer ip address

s: socket identifier

returns peer ip address

hn = socket.gethostbyaddr(s, addr)

gets the host name for ip address

s: socket identifier

addr: ip address

returns host name

ha = socket.gethostbyname(s, name)

gets the ip address for host name

s: socket identifier

name: host name

returns ip address

ha = socket.getsockname(s)

gets the socket name, where:

s: socket identifier

name: host name
returns socket name

ok = socket.send(s, data, flags)

sends data, where:
s: socket identifier
data: data to send
flags: optional flags
returns true on success

ok = socket.sendto(s, addr, port, data, flags)

sends data, where:
s: socket identifier
addr: address
port: port to send to
data: data to send
flags: optional flags
returns true on success

data = socket.recv(s, datasize, flags)

receives data, where:
s: socket identifier
datasize: data size to be received
flags: optional flags
returns received data

data = socket.recvfrom(s, addr, port, datasize, flags)

receives data from, where:
s: socket identifier
addr: address
port: port to receive from
datasize: data size to be received
flags: optional flags
returns received data

errf = socket.iserr(s)

```
-- BSD Socket
cls()
s = socket.new(socket.AF_INET, socket.SOCK_STREAM, socket.IPPROTO_TCP)
ip = socket.gethostbyname(s, "www.debian.org")
socket.connect(s, ip, 80)
socket.send(s, "HEAD / HTTP/1.0\r\n\r\n")
data = socket.recv(s, 1024)
print(data)
socket.delete(s)
```

Listing 17: Using BSD Sockets.

gets the error flag:

s: socket identifier

returns true if error flag set

errm = socket.geterr(s)

gets the error message:

s: socket identifier

Returns the error message, if any

socket.shutdown(s)

shut downs socket, where:

s: socket identifier

socket.delete(s)

delete socket and free resources, where:

s: socket identifier

An example of using sockets is given in **Listing 17**.

C Modules

In addition of the built-in modules, Solis includes some useful C modules:

lpeg pattern matching module with an example given in **Listing 18**.

lcrypt AES encryption module with an example given in **Listing 19**.


```
-- http://www.inf.puc-rio.br/~roberto/lpeg/  
local lpeg = require("lpeg")  
  
-- matches a word followed by end-of-string  
p = lpeg.R"az"^1 * -1  
  
print(p:match("hello"))      --> 6  
print(lpeg.match(p, "hello")) --> 6  
print(p:match("1 hello"))    --> nil
```

Listing 18: Pattern Matching Module.

```
lcrypt = require("lcrypt")  
  
function printbytes(t)  
  local str = '{ 0x'  
  for _,v in pairs(t) do  
    str = str .. string.format('%02X', v)  
  end  
  str = str .. ' }'  
  print(str, "\n")  
end  
  
cls()  
  
-- array of bytes  
inp = { 0x6B, 0xC1, 0xBE, 0xE2, 0x2E, 0x40, 0x9F, 0x96, 0xE9, 0x3D, 0x7E, 0x11,  
        0x73, 0x93, 0x17, 0x2A }  
  
-- AES key (16 bytes = 128 bits)  
key = { 0x2B, 0x7E, 0x15, 0x16, 0x28, 0xAE, 0xD2, 0xA6, 0xAB, 0xF7, 0x15, 0x88,  
        0x09, 0xCF, 0x4F, 0x3C }  
  
-- encrypt the array of bytes  
outp = lcrypt.encrypt(inp, key)  
  
-- decrypt the array of bytes to compare to the input data table  
inp_decrypted = lcrypt.decrypt(outp, key)  
printbytes(inp_decrypted)
```

Listing 19: AES Encryption Module.

```

-- Library for Arbitrary Precision Math (MAPM) by Michael C. Ring
-- Lua interface by Luiz Henrique de Figueiredo

lmapm = require ("lmapm")

-- lmapm library functions:
-- __add(x,y)      cbrt(x)          mod(x,y)
-- __div(x,y)      ceil(x)         mul(x,y)
-- __eq(x,y)       compare(x,y)    neg(x)
-- __lt(x,y)       cos(x)          number(x)
-- __mod(x,y)      cosh(x)         pow(x,y)
-- __mul(x,y)      digits([n])     round(x)
-- __pow(x,y)      digitisin(x)    sign(x)
-- __sub(x,y)      div(x,y)        sin(x)
-- __tostring(x)   exp(x)          sincos(x)
-- __unm(x)        exponent(x)     sinh(x)
-- abs(x)          factorial(x)     sqrt(x)
-- acos(x)         floor(x)        sub(x,y)
-- acosh(x)        idiv(x,y)       tan(x)
-- add(x,y)        inv(x)          tanh(x)
-- asin(x)         iseven(x)       tonumber(x)
-- asinh(x)        isint(x)        tostring(x,[n,exp])
-- atan(x)         isodd(x)        version
-- atan2(y,x)      log(x)
-- atanh(x)        log10(x)

print("\nSquare root of 2")
print("math.sqrt(2)  ", math.sqrt(2))
print("lmapm.sqrt(2) ", lmapm.sqrt(2))
print(lmapm.version)

```

Listing 20: Library for Arbitrary Precision Math (MAPM).

lmapm library for Arbitrary Precision Math (MAPM) by Michael C. Ring, modified Lua interface from **lmapm** by Luiz Henrique de Figueiredo. An example is given in **Listing 20**.

lxml library for XML using the **tinyxml** C++ library by Lee Thomason. An example is given in **Listing 21**.

```

-- Library for XML using the tinyxml C++ library by Lee Thomason

lxml = require ("lxml")
filename = "input.xml"
fp = io.open(filename, "r")
xmlcontent = fp:read("*all")
xmlvalue = lxml.read(xmlcontent, "solis", "device")
print("XML value of solis/device = " .. xmlvalue)

```

Listing 21: Library for XML.

User C modules: you can write your own C modules to use in Solis.

First, write your C module, as given in An example is given in **Listing 22**.

Second, build your C module with your favorite C compiler, Visual C++ on Windows or gcc or Clang on Linux, for example. Before compiling, add the Solis include directory (Solis Dir/include) in the compiler include path and set the linker to link against the Solis Lua Core library (libluacore.dll under Windows and libluacore.so under Linux). Under Visual C++, you can generate, for your specific VC version, the lib from libluacore.dll and libluacore.def by using the *dumpbin* and *lib* tools included in Visual C++ as shown in **Listing 23**.

Under Linux, you can use a Makefile such as in **Listing 24** (file located in *Solis/examples/miscellaneous/Cmodule*) and build with: `LD_RUN_PATH='$ORIGIN' make all`

A Visual C++ project example is given in *Solis/examples/miscellaneous/Cmodule*.

Of course, if your module is already built or purchased you can use it as usual with the Lua ad hoc functions.

Lastly, when your module was built as dll (under Windows) or so (under Linux), you can load it and use it under Solis by using the standard Lua functions as shown in **Listing 25**.

Instrumentation

In addition of the above presented modules, Solis includes C modules related to instrumentation using **visa** (for Windows only) and **lserial**.

lvisa module (for Windows only) to control instruments through GPIB, USB, Serial, Ethernet ... This module works with the interface VISA drivers (tested with NI and Agilent/Keysight GPIB cards. The drivers can be freely downloaded from the company site). Using lvisa is straightforward: load module with *require*, initialize, open visa connection, communicate and finally close connection.

Example on using VISA is given in **Listing 26**.

lserial module to control instruments through the serial port.

This module is available under Windows and Linux.

The example in **Listing 27** shows how to use the **lserial** module to communicate with an instrument.

Math Parser

The parser namespace includes functions to evaluate mathematical expressions:

```
p = parser.new()
```

```

/* put the Solis include dir in your include path */
#include <lua.h>
#include <lualib.h>
#include <luaXlib.h>

static void register_lualib(lua_State *L, const char *modulename, const luaL_Reg
    *funcs)
{
    #if LUA_VERSION_NUM < 502
        luaL_register(L, modulename, funcs);
    #else
        luaL_get_env(L);
        if (lua_istable(L, -1))
            luaL_setfuncs(L, funcs, 0);
        else {
            if (!lua_isnil(L, -1))
                luaL_error(L, "name conflict for module \"%s\"", modulename);

            luaL_newlib(L, funcs);
            lua_pushvalue(L, -1);
            lua_setglobal(L, modulename);
        }
    #endif
}

#ifdef WIN32
#define CMODULE_API __declspec(dllexport)
#else
#define CMODULE_API
#endif

static int Cmodule_version(lua_State *pLua)
{
    lua_pushstring(pLua, "Cmodule v0.1");
    return 1;
}

static const luaL_Reg modulefuncs[] = {
    { "version", Cmodule_version },
    { NULL, NULL }
};

CMODULE_API int luaopen_Cmodule(lua_State *pLua)
{
    register_lualib(pLua, "Cmodule", modulefuncs);
    return 1;
}

```

Listing 22: C Module Template.

```

cd C:\Solis\bin\
dumpbin /exports libluacore.dll > ..\lib\libluacore.txt
cd ..\lib\
echo LIBRARY libluacore > libluacore.def
echo EXPORTS >> libluacore.def
for /f "skip=19 tokens=4" %A in (libluacore.txt) do @echo %A >> libluacore.def
lib /def:"libluacore.def" /out:"libluacore.lib" /machine:x64

```

Listing 23: Generate Lib File from libluacore.dll for Visual C++.

```

WORKDIR = 'pwd'

CC = gcc

INC =
CFLAGS = -Wall -funwind-tables

INC_RELEASE = -I../..../include $(INC)
CFLAGS_RELEASE = -O2 -fPIC -std=gnu99 -DUSE_LUAJIT -std=gnu99 $(CFLAGS)
LDFLAGS_RELEASE = -s -L../..../include -lluacore $(LDFLAGS)
OBJDIR_RELEASE = .
OUT_RELEASE = ./Cmodule.so
OBJ_RELEASE = ./Cmodule.o

all: release

clean: clean_release

before_release:
    test -d . || mkdir -p .

after_release:

release: before_release out_release after_release

out_release: before_release $(OBJ_RELEASE)
    $(LD) -shared $(OBJ_RELEASE) -o $(OUT_RELEASE) $(LDFLAGS_RELEASE)

$(OBJDIR_RELEASE)/Cmodule.o: Cmodule.c
    $(CC) $(CFLAGS_RELEASE) $(INC_RELEASE) -c Cmodule.c -o $(OBJDIR_RELEASE)/
    Cmodule.o

clean_release:
    rm -f $(OBJ_RELEASE) $(OUT_RELEASE)

.PHONY: before_release after_release clean_release

```

Listing 24: Module Makefile.

```

-- Load C module, the Lua standard way
cls()
local Cmodule = require("Cmodule")
print(Cmodule.version())

```

Listing 25: Loading a C Module.

```

cls()
local vi = require("lvisa")           -- load the lvisa module
vi.load("visa32.dll")                -- load the driver DLL
v = vi.open("GPIB::8::INSTR")        -- open GPIB connection
vi.write(v, "*IDN?")                 -- send command to device
time.sleep(50)                       -- sleep during 50 ms
r,n = vi.read(v,100)                 -- read device IDN
print(r)                              -- print it
vi.close(v)                          -- close connection
print(vi.status())                   -- print status

```

Listing 26: VISA Example.

```
-- load the serial C module
local lserial = require("lserial")

-- open the serial port and configure it
-- port_handle = lserial.open(port_num, settings, verbose)
-- port_num: usually 1 (COM1) or 2 (COM2)
-- settings has the same meaning than in the Windows BuildCommDCB
-- function except that BuildCommDCB has no 'timeout' parameter
-- that is specific to this Solis module
-- verbose (optional): true to show detailed messages
-- returns the port handle
p = lserial.open(1, "baud=4800 parity=n data=8 stop=2 timeout=1000", true)

-- write to the serial port
-- lserial.write(port_handle, command)
-- port_handle: the port handle, as returned by lserial.open
-- command to send to the device
-- returns the number of bytes written
lserial.write(p, "*IDN?\n")

-- read from the serial port
-- r,n = lserial.read(port_handle, bytes)
-- port_handle: the port handle, as returned by lserial.open
-- bytes: number of bytes to read
-- returns the string read and the number of bytes read
r,n = lserial.read(p, 128)
io.write("\n recv = ", r, " ; bytes read = ", n)

-- close the serial port
-- lserial.close(port_handle)
-- port_handle: the port handle, as returned by lserial.open
lserial.close(p)
```

Listing 27: RS232 Example.

creates a new parser.

parser.set(p, name, value)

sets variable.

val = parser.get(p, name)

returns variable value.

val = parser.eval(p, expr)

evaluates math expression.

val = parser.evalf(p, func, x)

evaluates math function for x value.

val = parser.solve(p, eq, a, b)

solves equation in [a,b] interval.

parser.delete(p)

deletes a parser.

The math parser supports the following functions:

Exp(x)

exponential

Ln(x)

natural (Napierian) logarithm

Log(x)

decimal logarithm

Log2(x)

base-2 logarithm

Sin(x)

sine

Cos(x)

cosine

Tan(x)

tangent

Asin(x)

arc sine

Acos(x)

arc cosine

Atan(x)

arc tangent

Sinh(x)

hyperbolic sine

Cosh(x)

hyperbolic cosine

Tanh(x)

hyperbolic tangent

Abs(x)

absolute value

Sqrt(x)

square root

Cbrt(x)

cubic root

Ceil(x)

ceiling, the smallest integer not less than x

Floor(x)

integer part of x

Rand()

random number between 0 and 1

Sign(x)

sign of x (-1 if $x < 0$, +1 if $x > 0$ and 0 if $x = 0$)

Erf(x)

error function

Fact(x)

factorial of x

The math parser supports also the following constants:

Pi**e**

Natural (Napierian) logarithm base (2.71828...)

Universal constants in international units (SI)

q

Electron charge (in C)

me

Electron mass (kg)

mp

Proton mass (kg)

kB

Boltzmann constant (J/K)

h

Planck constant (Js)

c

Speed of Light in vacuum (m/s)

eps0

```

-- Parser
cls()
p = parser.new()
parser.set(p, "x", 1)
parser.set(p, "a", 2)
y = parser.eval(p, "a*x + sin(x/a) + 2")
print("y = " .. y)

```

Listing 28: Using the Math Parser.

Electric constant (F/m)

mu0

Magnetic constant (N/A²)

NA

Avogadro constant (1/mole)

G

Constant of gravitation (m³/kg/s²)

Ri

Rydberg constant (1/m)

F

Faraday constant (C/m)

R

Molar gas constant (J/mole/K)

Example of using the parser module is given in **Listing 28**.

Graphical User Interface using IUP

Solis includes the IUP toolkit for building graphical user interfaces through the `iuplua` namespace. The documentation and example of using `iuplua` can be found at the IUP website: <https://www.tecgraf.puc-rio.br/iup/> and a quickstart guide here: <https://www.tecgraf.puc-rio.br/iup/en/basic/>. Note that the other IUP lua extension modules, such as `iupluacontrols`, `cd` or `iuplua_plot51`, are not included in Solis. Only the main IUP lua module `iuplua` is included.

An example of using `iuplua` is given in **Listing 29**.

```
-- https://www.tecgraf.puc-rio.br/iup/en/basic/index.html

require "iuplua"

counter = 0
text = iup.text{readonly = "YES", value = "", expand = "YES", alignment = "
    ACENTER"}
button = iup.button{title = "Stop", expand = "YES"}

function idle_cb()
    counter = counter + 1
    text.value = string.format("Iteration %d", counter)
    if counter == 10000 then
        iup.SetIdle(nil)
        button.title = "Start"
    end
    return iup.DEFAULT
end

function button:action()
    text.value = ""
    counter = 0
    iup.SetIdle(nil)
    if button.title == "Stop" then
        button.title = "Start"
    else
        iup.SetIdle(idle_cb)
        button.title = "Stop"
    end
end

dlg = iup.dialog
{
    iup.vbox
    {
        iup.hbox
        {
            button;
            margin = "20x20",
            alignment = "ACENTER"
        };
        iup.hbox
        {
            text;
            margin = "20x20",
            alignment = "ACENTER"
        };
    },
    title = "IUP",
    size = "160x80",
    icon = 0,
}

dlg:showxy(iup.CENTER, iup.CENTER)
iup.SetIdle(idle_cb)
if (iup.MainLoopLevel() == 0) then
    iup.MainLoop()
end
iup.Close()
```

Listing 29: Graphical User Interface using IUP.

Solis Calculator

The Solis scientific calculator, `SolisCalc`, is an advanced mathematical expression-based calculator (screenshot in figure ??). It supports the most common and useful functions. It's easy to use: to evaluate an expression, simply write it, using operators (`+` `-` `*` `/` `^`), parenthesis and mathematical functions and press ENTER (or F12). One can also use the numeric keypad to enter numbers and operators. One can set variables (with any non-reserved name), using fundamental constants, etc. `SolisCalc` menu gives an easy way to use the software functionality.

The supported mathematical **functions**, **constants** and **commands** are shown in **Listing 30**, **Listing 31** and **Listing 32**.

You can create **variables** (with any non-reserved name): `a = 6*2`

The last expression evaluation can be accessed using the internal variable `ans`: `ans`

Append a semicolon (`;`) to the expression to suppress the output: `a = 6*2;`

A comment can be added at the end of an expression, using `#` : `y=sin(pi/4) # comment`

Previous calculated expressions can be reused by pressing up or down arrows.

`SolisCalc` can be executed from the **command line**: `soliscalc -run input [-out outfile]`

input may be a filename or a double-quoted expression and `outfile` is the output filename:

```
soliscalc -run "a=1;b=2;c=a*sin(b)"
soliscalc -run calcin.txt -out calcout.txt
```

Integer arithmetic in binary, octal and hexadecimal bases:

One can perform integer calculations in binary, octal, decimal and hexadecimal bases in 32 bits unsigned format directly within the console, by prefixing the number with `0b` for binary base, `0o` for octal base and `0x` for hexadecimal base: just type the expression and press enter: `0xFFFF7 +`

```

exp(x)           # exponential
ln(x)            # natural logarithm
log(x)           # decimal logarithm
pow(x,n)         # xn
sin(x)           # sine
cos(x)           # cosine
tan(x)           # tangent
asin(x)          # arc sine
acos(x)          # arc cosine
atan(x)          # arc tangent
sinh(x)          # hyperbolic sine
cosh(x)          # hyperbolic cosine
tanh(x)          # hyperbolic tangent
abs(x)           # absolute value
sqrt(x)          # square root
ceil(x)          # ceiling, the smallest integer not less than x
floor(x)         # floor, the largest integer not greater than x
fmod(x,y)        # x modulo y
erf(x)           # error function
jn(n,x)          # Bessel function of x of the first kind of order n
yn(n,x)          # Bessel function of x of the second kind of order n
bern(x)          # Bernoulli function: x / (exp(x) - 1)
gauss(x,m,s)     # Gauss function: exp((x - m)^2 / 2s^2)
lorentz(x,m,s)   # Lorentz function: s / ((x - m)^2 + s^2)
hypot(x,y)       # hypotenuse, sqrt(x^2 + y^2)
min(x,y)         # smallest value of x and y
max(x,y)         # largest value of x and y
rand(x)          # random number between 0 and 1
time()           # elapsed time in seconds since January 1, 1970
sign(x)          # sign of x (-1 if x < 0, +1 if x > 0 and 0 if x = 0)
hypot(x,y)       # sqrt(x^2+y^2)
erf(x)           # error function
gammaln(x)       # ln(gamma(x))
beta(x)          # beta(x)
trunc(x)         # nearest integer
round(x)         # nearest integer, rounding
rint(x)          # rounds the floating-point to an integer

```

Listing 30: Mathematical Functions in SolisCalc.

```

pi              # 3.14169...
_q              # electron charge in C
_m              # electron mass in kg
_k              # Boltzmann constant in J / K
_h              # Planck constant in m2 kg / s
_c              # speed of light in vacuum in m / s
_e              # vacuum permittivity in F / m
_n              # Avogadro constant

```

Listing 31: Constants in SolisCalc.

```
cls                # clear the console
format:real        # set the numerical output format to real
format:int         # set the numerical output format to integer (10-base)
format:dec         # set the numerical output format to integer (10-base)
format:bin         # set the numerical output format to integer (2-base)
format:oct         # set the numerical output format to integer (8-base)
format:hex         # set the numerical output format to integer (16-base)
format?           # print out the current numerical format
unit:degree        # set the angle unit to degree
unit:radian        # set the angle unit to radian
unit?             # print out the angle unit
```

Listing 32: Commands in SolisCalc.

12 + 0b111 + 0o547

To show result in hexadecimal, set accordingly the numerical output format (see **Listing 32**):

```
format:hex
```

To solve a nonlinear equation $f(x) = 0$: `solve(x2 - 2, 0, 10)`

The syntax is `solve(function, x1, xh)` where `function` is the $f(x)$ function expression (example: $x^2 - 2$), `x1` the x lower limit of the interval where the solution is to be found and `xh` the x higher limit (`x1` and `xh` are optional).

A session (calculations history) can be saved in a text file (with extension `.soliscalc`) and retrieved later: menu *File/Save* and *File/Open*.

Bibliography

- [1] S Ierusalimschy, L H Figueiredo, and Celes W. *Lua Reference Manual*. Lua.org, 2020 (cited p. 1).
- [2] *The IUP GUI toolkit, (C) 1994-2017 Tecgraf/PUC-Rio*. 2017 (cited p. 1).
- [3] S Ould Saad Hamady. “Solis: a modular, portable, and high-performance 1D semiconductor device simulator”. In: *Journal of Computational Electronics* (2020), pp. 1–8 (cited p. 1).
- [4] S Ould Saad Hamady. “SigmaSim: Enhanced One-Dimensional Simulator for Semiconductor Devices”. In: *37th International Symposium on Compound Semiconductors (ISCS), Takamatsu, Japan. Oral*. 2010 (cited p. 2).