The PracTEX Journal, 2013, No. 1
Article revision 2013/1/27

Numerical methods with LuaLATEX

Juan I. Montijano, Mario Pérez, Luis Rández and Juan Luis Varona

Email {monti,mperez,randez}@unizar.es and jvarona@unirioja.es
Address Universidad de Zaragoza (Zaragoza, Spain) and Universidad de La Rioja (Logroño, Spain)

Abstract An extension of TEX known as LuaTEX has been in development for the past few years. Its goal is to allow TEX to execute scripts written in the general purpose programming language called Lua. There is also LuaLATEX, which is the corresponding extension for LATEX.

In this paper, we show how LuaLATEX can be used to perform tasks that require a large amount of mathematical computation. With LuaLATEX instead of LATEX, we achieve important improvements: since Lua is a general purpose language, rendering documents that include evaluation of mathematical algorithms is much easier, and generating the pdf file becomes much faster.

Introduction

TEX (and LATEX) is a document markup language used to typeset beautiful papers and books. Although it can also do programming commands such as conditional execution, it is not a general purpose programming language. Thus there are many tasks that are easily done with other programming languages, but are very complicated or very slow when done with TEX. Due to this limitation, auxiliary programs have been developed to assist TEX with common tasks related to document preparation. For instance, bibtex or biber to build bibliographies, and makeindex or xindy to generate indexes. In both cases, sorting a list alphabetically is a relatively simple task for most programming languages, but it is very complicated to do with TEX, which is why auxiliary applications written in standard programming languages are used to prepare the bibliography and index.

Another shortcoming of TEX is the computation of mathematical expressions. One of the most common uses of TEX is to compose mathematical formulas, and it does this extremely well. However TEX is not good at computing mathematics.

For instance, TEX itself does not have built-in functions to compute a square root, a sine or a cosine. It is possible to compute mathematical functions with the help of auxiliary packages written in TEX but, internally, these packages must compute functions using only addition, subtraction, multiplication and division; and a large amount of them. This is difficult to program (for package developers) and slow in the execution.

To address the need to do more complex functions within TEX, an extension of TEX called LuaTEX was begun a few years ago. (The leaders of the project and main developers are Taco Hoekwater, Hartmut Henkel and Hans Hagen.) The idea was to enhance TEX with a previously existing general purpose programming language. After a careful evaluation of possible candidates, the language chosen was Lua (see `http://www.lua.org/`), a powerful, fast, lightweight, embeddable scripting language that has, of course, a free license suitable to be used with TEX. Moreover, Lua is easy to learn and use, and anyone with basic programming skills can use it without difficulty. (Many examples of Lua code can be found later in this article, and also in `http://rosettacode.org/wiki/Category:Lua`, and `http://lua-users.org/`).

LuaTEX is not TEX, but an extension of TEX, in the same way that pdfTEX or XƎTEX are also extensions. In fact, LuaTEX includes pdfTEX (it is an extension of pdfTEX, and offers backward compatibility), and also has many of the features of XƎTEX.

LuaTEX is still in a beta stage, but the current versions are usable (the first public beta was launched in 2007, and when this paper was written in January 2013, the release used was version 0.74).

It has many new features useful for typographic composition, and examples can be seen at the project web site `http://www.luatex.org`, and some papers using development versions have been published in TUGboat, among them [1, 2, 3, 4, 6, 8]. Most of articles are devoted to the internals and are very technical, only for true TEX wizards; we do not deal with this in this paper. Instead, our goal is to show how the mathematical power of the embedded language Lua can be used in LuaTEX. Of course, when we build LATEX over LuaTEX, we get the so-called LuaLATEX, which is more familiar for regular LATEX users.

All the examples in this paper are done with LuaLATEX. It is important to note that the current version of LuaTEX is not meant for production and beta users are warned of possible future changes in the syntax. For the examples in this article we use only a few general Lua-specific commands, so it is likely these examples will also work in future versions.

To process a LuaLaTeX document we perform the following steps: First, we must compile with LuaLaTeX, not with LaTeX; how to do this depends on which editor we are using. Second, we must load the package luacode with `\usepackage{luacode}`. Then, inside LuaLaTeX, we jump into Lua mode with the command `\directlua`; moreover, we can define Lua routines in a `\begin{luacode}` – `\end{luacode}` environment (also `{luacode*}` instead of `{luacode}` can be used); the precise syntax can be found in the manual "The luacode package" (by Manuel Pégourié-Gonnard). In the examples, we do not explain all the details of the code; they are left to the reader's intuition.

In this paper we present four examples. The first is very simple: the computation of a trigonometric table. In the other examples we use the LaTeX packages tikz and pgfplots to show Lua's ability to produce graphical output. Some mathematical skill may be necessary to fully understand the examples, but the reader can nevertheless see how Lua is able to manage the computation-intensive job. In any case, we do not explore the more complex possibilities, which involve writing Lua programs that load existing Lua modules or libraries to perform a wide range of functions and specialized tasks.

1 First example: a trigonometric table

To show how to use Lua, let us begin with a simple but complete example. Observe the following source code. Typesetting it with LuaLaTeX, we get the trigonometric table shown in Figure 1.

```
\documentclass{article}
\usepackage{luacode}

\begin{luacode*}
  function trigtable ()
    for t=0, 45, 3 do
      x=math.rad(t)
      tex.print(string.format(
      '%2d$^{\\circ}$ & %1.9f & %1.9f & %1.9f & %1.9f \\\\',
      t, x, math.sin(x), math.cos(x), math.tan(x)))
    end
  end
\end{luacode*}
\newcommand{\trigtable}{\luadirect{trigtable()}}

\begin{document}
\begin{tabular}{rcccc}
  \hline
  & $x$ & $\sin(x)$ & $\cos(x)$ & $\tan(x)$ \\
  \hline
  \trigtable
  \hline
\end{tabular}
\end{document}
```

The `luacode*` environment contains a small Lua program with a function named `trigtable` (without arguments). This function consists of a loop with a variable `t` representing degrees. Lua converts `t` to radians with `x=math.rad(t)`; then, Lua computes the sine, the cosine and the tangent. Inside Lua mode, it "exports" to LaTeX with `tex.print`; note that we escape any backslash by doubling it. Moreover, we have taken into account the following notation to give format to numbers:

- `%2d` indicates that a integer number must be displayed with 2 digits.
- `%1.9f` indicates that a floating point number must be displayed with 1 digit before the decimal point and 9 digits after it.

The LaTeX part has the skeleton of a tabular built with the data exported by Lua.

2 Second example: Gibbs phenomenon

Now and in what follows, we are using graphics to show the output of some mathematical routines. A very convenient way to do it is by means of the PGF/Ti*k*Z package (Ti*k*Z is a a high-level interface to PGF) by Till Tantau (the huge manual of

	x	$\sin(x)$	$\cos(x)$	$\tan(x)$
0°	0.000000000	0.000000000	1.000000000	0.000000000
3°	0.052359878	0.052335956	0.998629535	0.052407779
6°	0.104719755	0.104528463	0.994521895	0.105104235
9°	0.157079633	0.156434465	0.987688341	0.158384440
12°	0.209439510	0.207911691	0.978147601	0.212556562
15°	0.261799388	0.258819045	0.965925826	0.267949192
18°	0.314159265	0.309016994	0.951056516	0.324919696
21°	0.366519143	0.358367950	0.933580426	0.383864035
24°	0.418879020	0.406736643	0.913545458	0.445228685
27°	0.471238898	0.453990500	0.891006524	0.509525449
30°	0.523598776	0.500000000	0.866025404	0.577350269
33°	0.575958653	0.544639035	0.838670568	0.649407593
36°	0.628318531	0.587785252	0.809016994	0.726542528
39°	0.680678408	0.629320391	0.777145961	0.809784033
42°	0.733038286	0.669130606	0.743144825	0.900404044
45°	0.785398163	0.707106781	0.707106781	1.000000000

Figure 1: A trigonometric table.

the current version 2.10 has more than 700 pages of complete documentation and examples); a nice and small paper with a introduction is [7]. Based on PGF/Ti*k*Z, the package PGFPLOTS (by Christian Feuersänger) has additional facilities to plot mathematical functions like $y = f(x)$ (or a parametric function $x = f(t)$, $y = g(t)$) or visualize data in two or three dimensions. For instance, PGFPLOTS can draw the axis automatically, as usual in any graphic software.

For completeness, let us start showing the syntax of PGFPLOTS by means of a data plot; this is an example extracted from its very complete manual (more than 400 pages in the present version 1.7). After loading `\usepackage{pgfplots}`, the code

```
\begin{tikzpicture}
\begin{axis}[xlabel=Cost, ylabel=Error]
  \addplot[color=red,mark=x] coordinates {
    (2,-2.8559703) (3,-3.5301677) (4,-4.3050655)
    (5,-5.1413136) (6,-6.0322865) (7,-6.9675052)
```

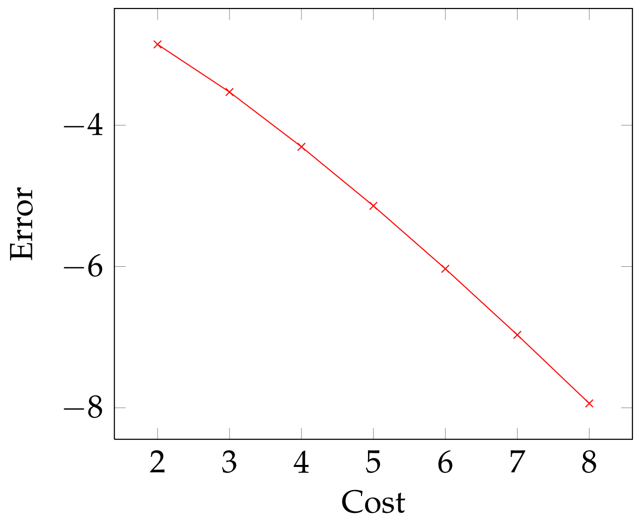

Figure 2: Plotting of a data table with the PGFPLOTS package.

```
    (8,-7.9377747)
  };
\end{axis}
\end{tikzpicture}
```

provides the plot in Figure 2. Before going on, note that in future versions the packages PGF/Ti*k*Z and PGFPLOTS could, internally, use LuaLaTeX themselves in a way transparent to the user. This will allow extra power, calculating speed, and simplicity, but this is not yet available and we will not worry about it in this paper.

In the next example we consider the Gibbs phenomenon. Using LuaLaTeX, the idea is to compute a data table with Lua (that is easy to program, powerful and fast in the execution), and plot it with PGFPLOTS.

The Gibbs phenomenon is the peculiar way in which the Fourier series of a piecewise continuously differentiable periodic function behaves at a jump discontinuity, where the n-th partial sum of the Fourier series has large oscillations near the jump. It is explained in many harmonic analysis texts, but for the purpose of this paper the reader can refer to [9].

In our case we consider the function $f(x) = (\pi - x)/2$ in the interval $(0, 2\pi)$ extended by periodicity to the whole real line (it has discontinuity jumps at $2j\pi$

for every integer j). Its Fourier series is

$$f(x) = \sum_{k=1}^{\infty} \frac{\sin(kx)}{k}.$$

To show the Gibbs phenomenon, we evaluate the partial sum $\sum_{k=1}^{n} \frac{\sin(kx)}{k}$ (for $n = 30$) with Lua to generate a table of data, and we plot it with PGFPLOTS.

In the `.tex` file, we include the following Lua part to compute the partial sum (function `partial_sum`) and to export the data with the syntax required by PGFPLOTS (function `print_partial_sum`):

```
\begin{luacode*}
-- Fourier series
function partial_sum(n,x)
    partial = 0;
    for k = 1, n, 1 do
        partial = partial + math.sin(k*x)/k
    end;
    return partial
end

-- Code to write PGFplots data as coordinates
function print_partial_sum(n,xMin,xMax,npoints,option)
    local delta = (xMax-xMin)/(npoints-1)
    local x = xMin
    if option~=[[]] then
        tex.sprint("\\addplot["..option.."] coordinates{")
    else
        tex.sprint("\\addplot coordinates{")
    end
    for i=1, npoints do
        y = partial_sum(n,x)
        tex.sprint("("..x..","..y..")")
        x = x+delta
    end
    tex.sprint("}")
```

```
end
\end{luacode*}
```

Then, we also define the command

```
\newcommand\addLUADEDplot[5][]{%
  \directlua{print_partial_sum(#2,#3,#4,#5,[[#1]])}%
}
```

which will be used to call the data from PGFPLOTS. Here, the parameters have the following meaning: `#2` indicates the number of terms to be added ($n = 30$ in our case); the plot will be done in the interval [`#3`, `#4`] (from $x = 0$ to 10π) sampled in `#5` points (to get a very smooth graphic and to show the power of the method we use 1000 points); finally, the optional argument `#1` is used to manage optional arguments in the `\addplot` environment (for instance color, width of the line, ...).

Now, the plot is generated by

```
\pgfplotsset{width=15cm, height=7cm}
\begin{tikzpicture}\small
\begin{axis}[xmin=-0.2, xmax=31.6, ymin=-1.85, ymax=1.85,
  xtick={0,5,10,15,20,25,30},
  ytick={-1.5,-1.0,-0.5,0.5,1.0,1.5},
  minor x tick num=4,
  minor y tick num=4,
  axis lines=middle,
  axis line style={-}
  ]
% SYNTAX: Partial sum 30, from x = 0 to 10*pi, sampled in 1000 points
\addLUADEDplot[color=blue,smooth]{30}{0}{10*math.pi}{1000};
\end{axis}
\end{tikzpicture}
```

See the output in Figure 3.

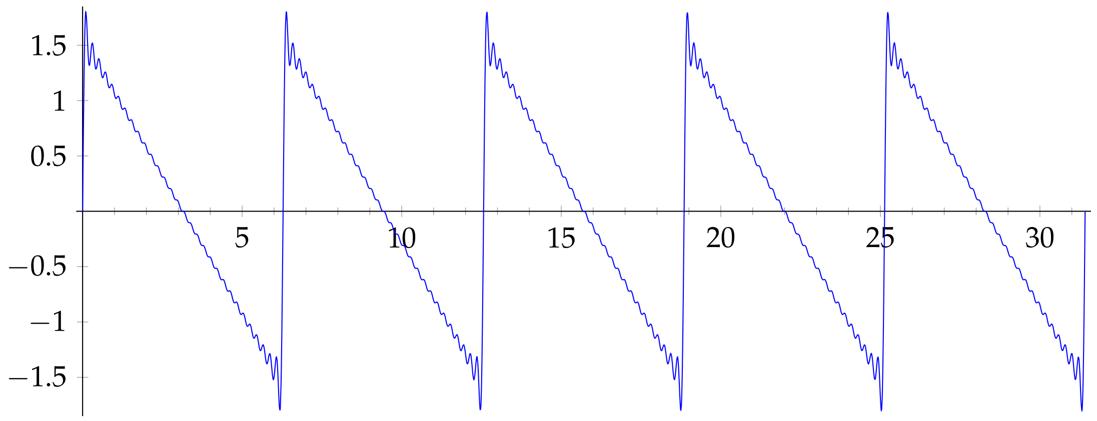

Figure 3: The partial sum $\sum_{k=1}^{30} \frac{\sin(kx)}{k}$ of the Fourier series of $f(x) = (\pi - x)/2$ illustrating the Gibbs phenomenon.

3 Third example: Runge-Kutta method

A differential equation is an equation that links an unknown function and its derivatives, and these equations play a prominent role in engineering, physics, economics, and other disciplines. When the value of the function at an initial point is fixed, a differential equation is known as an initial value problem. The mathematical theory of differential equations shows that, under very general conditions, an initial value problem has a unique solution. Usually, it is not possible to find the exact solution in an explicit form, and it is necessary to approximate it by means of numerical methods.

One of the most popular methods to integrate numerically an initial value problem

$$\begin{cases} y'(t) = f(t, y(t)), \\ y(t_0) = y_0 \end{cases}$$

is the *classical* Runge-Kutta method of order 4. With it, we compute in an approximate way the values $y_i \simeq y(t_i)$ at a set of points $\{t_i\}$ starting from $i = 0$ and with $t_{i+1} = t_i + h$ for every i by the algorithm

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4),$$

where

$$\begin{cases} k_1 = hf(t_i, y_i), \\ k_2 = hf(t_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1), \\ k_3 = hf(t_i + \frac{1}{2}h, y_i + \frac{1}{2}k_2), \\ k_4 = hf(t_i + h, y_i + k_3). \end{cases}$$

See, for instance, [11].

Here, we consider the initial value problem

$$\begin{cases} y'(t) = y(t)\cos\big(t + \sqrt{1 + y(t)}\big), \\ y(0) = 1. \end{cases}$$

In the Lua part of our `.tex` file, we compute the values $\{(t_i, y_i)\}$ and export them with PGFPLOTS syntax by means of

```
\begin{luacode*}
-- Differential equation y'(t) = f(t,y)
-- with f(t,y) = y * cos(t+sqrt(1+y)).
-- Initial condition: y(0) = 1
function f(t,y)
    return y * math.cos(t+math.sqrt(1+y))
end

-- Code to write PGFplots data as coordinates
function print_RKfour(tMax,npoints,option)
    local t0 = 0.0
    local y0 = 1.0
    local h = (tMax-t0)/(npoints-1)
    local t = t0
    local y = y0
    if option~=[[]] then
        tex.sprint("\\addplot["..option.."] coordinates{")
    else
        tex.sprint("\\addplot coordinates{")
    end
    tex.sprint("("..t0..","..y0..")")
    for i=1, npoints do
        k1 = h * f(t,y)
        k2 = h * f(t+h/2,y+k1/2)
        k3 = h * f(t+h/2,y+k2/2)
        k4 = h * f(t+h,y+k3)
        y = y + (k1+2*k2+2*k3+k4)/6
        t = t + h
        tex.sprint("("..t..","..y..")")
    end
    tex.sprint("}")
end
\end{luacode*}
```

Also, we define the command

```
\newcommand\addLUADEDplot[3][]{%
  \directlua{print_RKfour(#2,#3,[[#1]])}%
}
```

to call the Lua routine from the LaTeX part (the parameter #2 indicates the final value of *t*, and #3 is the number of sampled points).

Then, the graphic of Figure 4, which shows the solution of our initial value problem, is done by means of

```
\pgfplotsset{width=0.9\textwidth, height=0.6\textwidth}
\begin{tikzpicture}
\begin{axis}[xmin=-0.5, xmax=30.5, ymin=-0.02, ymax=1.03,
  xtick={0,5,...,30}, ytick={0,0.2,...,1.0},
  enlarge x limits=true,
  minor x tick num=4, minor y tick num=4,
  axis lines=middle, axis line style={-}
  ]
% SYNTAX: Solution of the initial value problem
% in the interval [0,30] sampled at 200 points
\addLUADEDplot[color=blue,smooth]{30}{200};
\end{axis}
\end{tikzpicture}
```

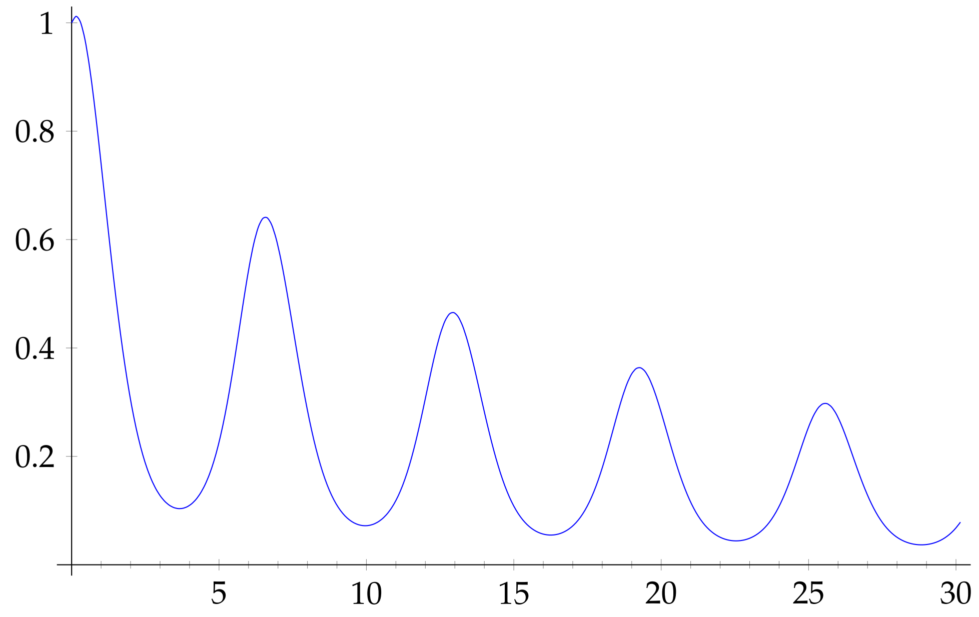

Figure 4: Solution of the differential equation $y'(t) = y(t)\cos\big(t + \sqrt{1+y(t)}\big)$ with initial condition $y(0) = 1$.

4 Fourth example: Lorenz attractor

The Lorenz attractor is a strange attractor that arises in a system of equations describing the 2-dimensional flow of a fluid of uniform depth, with an imposed vertical temperature difference. In the early 1960s, Lorenz [5] discovered the chaotic behavior of a simplified 3-dimensional system of this problem, now known as the Lorenz equations:

$$\begin{cases} x'(t) = \sigma(y(t) - x(t)), \\ y'(t) = -x(t)z(t) + \rho x(t) - y(t), \\ z'(t) = x(t)y(t) - \beta z(t). \end{cases}$$

The parameters σ, ρ, and β are usually assumed to be positive. Lorenz used the values $\sigma = 10$, $\rho = 28$ and $\beta = 8/3$. The system exhibits a chaotic behavior for these values; actually, it became the first example of a chaotic system. A more complete description can be found in [10].

Figure 5 shows the numerical solution of the Lorenz equations calculated with $\sigma = 3$, $\rho = 26.5$ and $\beta = 1$. Six orbits starting at several initial points close to

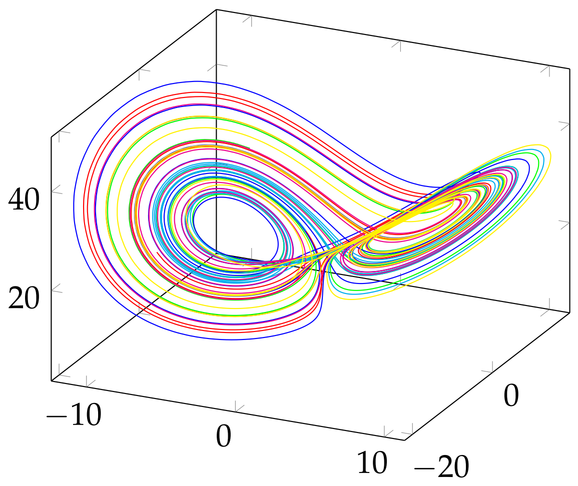

Figure 5: The Lorentz attractor (six orbits starting at several initial points).

$(0,1,0)$ are plotted in different colors; all of them converge to the 3-dimensional chaotic attractor known as the Lorenz attractor.

The Lua part of the program uses a discretization of the Lorenz equations (technically, this is the explicit Euler method $y_{i+1} = y_i + hf(t_i, y_i)$, which is less precise than the Runge-Kutta method of the previous section, but enough to find the attractor):

```
\begin{luacode*}
-- Differential equation of the Lorenz attractor
function f(x,y,z)
    local sigma = 3
    local rho = 26.5
    local beta = 1
    return {sigma*(y-x), -x*z + rho*x - y, x*y - beta*z}
end

-- Code to write PGFplots data as coordinates
function print_LorAttrWithEulerMethod(h,npoints,option)
    -- The initial point (x0,y0,z0)
    local x0 = 0.0
    local y0 = 1.0
    local z0 = 0.0
    -- we add a random number between -0.25 and 0.25
    local x = x0 + (math.random()-0.5)/2
    local y = y0 + (math.random()-0.5)/2
    local z = z0 + (math.random()-0.5)/2
    if option~=[[]] then
        tex.sprint("\\addplot3["..option.."] coordinates{")
    else
        tex.sprint("\\addplot3 coordinates{")
    end
    -- we dismiss the first 100 points to go into the attractor
    for i=1, 100 do
        m = f(x,y,z)
        x = x + h * m[1]
        y = y + h * m[2]
        z = z + h * m[3]
    end
    for i=1, npoints do
        m = f(x,y,z)
        x = x + h * m[1]
        y = y + h * m[2]
        z = z + h * m[3]
        tex.sprint("("..x..","..y..","..z..")")
    end
    tex.sprint("}")
end
\end{luacode*}
```

The function which calls the Lua part from the LaTeX part is

```
\newcommand\addLUADEDplot[3][]{%
  \directlua{print_LorAttrWithEulerMethod(#2,#3,[[#1]])}%
}
```

Here, the parameter #2 gives the step of the discretization, and #3 is the number of points.

The LaTeX part is the following. In it, we call the Lua function six times with different colors:

```
\begin{tikzpicture}
\begin{axis}
% SYNTAX: Solution of the Lorenz system
% with step h=0.02 sampled at 1000 points.
  \addLUADEDplot[color=red,smooth]{0.02}{1000};
  \addLUADEDplot[color=green,smooth]{0.02}{1000};
  \addLUADEDplot[color=blue,smooth]{0.02}{1000};
  \addLUADEDplot[color=cyan,smooth]{0.02}{1000};
  \addLUADEDplot[color=magenta,smooth]{0.02}{1000};
  \addLUADEDplot[color=yellow,smooth]{0.02}{1000};
\end{axis}
\end{tikzpicture}
```

References

[1] T. Hoekwater and H. Henkel, *LuaTEX 0.60: An overview of changes*, TUGboat, Volume 31 (2010), No. 2, 174–177. Available from https://www.tug.org/TUGboat/tb31-2/tb98hoekwater.pdf

[2] H. Hagen, *LuaTEX: Halfway to version 1*, TUGboat, Volume 30 (2009), No. 2, 183–186. Available from https://www.tug.org/TUGboat/tb30-2/tb95hagen-luatex.pdf

[3] P. Isambert, *Three things you can do with LuaTEX that would be extremely painful otherwise*, TUGboat, Volume 31 (2010), No. 3, 184–190. Available from https://www.tug.org/TUGboat/tb31-3/tb99isambert.pdf

[4] P. Isambert, *OpenType fonts in LuaTEX*, TUGboat, Volume 33 (2012), No. 1, 59–85. Available from https://www.tug.org/members/TUGboat/tb33-1/tb103isambert.pdf

[5] E. N. Lorenz, *Deterministic Nonperiodic Flow*, J. Atmospheric Sci., Volume 20 (1963), No. 2, 130–141. Available from http://dx.doi.org/10.1175/1520-0469(1963)020<0130:DNF>2.0.CO;2

[6] A. Mahajan, *LuaTEX: A user's perspective*, TUGboat, Volume 30 (2009), No. 2, 247–251. Available from https://www.tug.org/TUGboat/tb30-2/tb95mahajan-luatex.pdf

[7] A. Mertz and W. Slough, *Graphics with PGF and TikZ*, The PracTEX Journal, 2007, No. 1. Available from http://www.tug.org/pracjourn/2007-1/mertz/

[8] A. Reutenauer, *LuaTEX for the LATEX user: An introduction*, TUGboat, Volume 30 (2009), No. 2, 169. Available from https://www.tug.org/TUGboat/tb30-2/tb95reutenauer.pdf

[9] Wikipedia, *Gibbs phenomenon*. Available from http://en.wikipedia.org/wiki/Gibbs_phenomenon

[10] Wikipedia, *Lorenz system*. Available from http://en.wikipedia.org/wiki/Lorenz_system

[11] Wikipedia, *Runge-Kutta methods*. Available from http://en.wikipedia.org/wiki/Runge-Kutta_methods