

Gnuplot - Introduction

Dragos Chirila (cdragos@awi.de)

December 7, 2009

1 What is *Gnuplot*?

- General-purpose function and data plotting program;
- It is *open-source*;
- Offers an *interactive* mode (user types commands in the Gnuplot shell);
- Scripting capabilities (i.e. automatize the entire plotting procedure): *VERY* useful while debugging programs; main idea behind this feature: you will basically need to plot the same type & format of output data, while you perfect your algorithms.

2 Basic usage of *Gnuplot*

Simply typing:

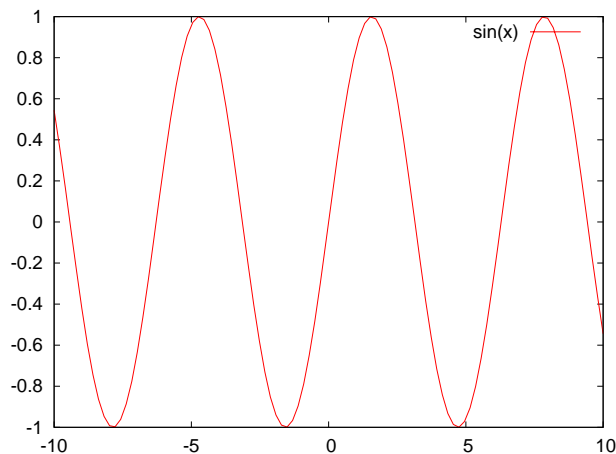
```
gnuplot
```

at a console will drop you in Gnuplot's *interactive mode*, with a cursor which looks like this:

```
gnuplot>
```

To quit the program, simply type **quit** in the *Gnuplot shell*. In the following, we will always show this cursor in the examples, to indicate that we are working in *interactive mode*. Now, you can already start plotting things. For example, try:

```
gnuplot> plot sin(x)
```



As you can see, it opened another window which shows the sine function. But often this is not good enough, as we may have further expectations from our plots, such as plotting in other ranges, changing the plotting style, label the axes, etc. We will show how to do some of these things in a few moments.

3 More about plotting

As you probably realized by now, *Gnuplot* is all about plotting. Hence, our small tutorial will gravitate around the *plot* and *splot* commands, which are *Gnuplot*'s instructions for *2D* (line plots: 1 set of values for the coordinate and 1 set of values for the function) and *3D* (surface plots: 2 set of values for the coordinates and 1 set of values for the function) plots respectively. The basic syntax for the commands is:

```
plot {<ranges>}
     {<function> | {"<datafile>" {datafile-modifiers}}}}
     {axes <axes>} {<title-spec>} {with <style>}
     {, {definitions,} <function> ...}
```

```
splot {<ranges>}
      <function> | "<datafile>" {datafile-modifiers}}
      {<title-spec>} {with <style>}
      {, {definitions,} <function> ...}
```

Well, the syntax by itself may seem a little cryptic in the beginning, so let us see what it means in practice:

3.1 The *plot* command:

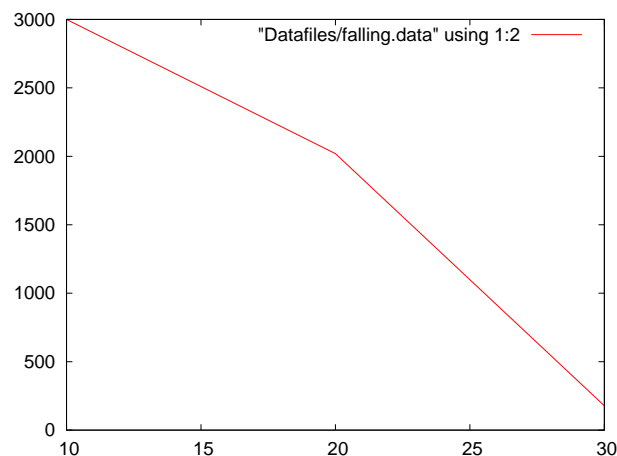
It can plot *functions* (as in the sine example above) or *datafiles*. While it is easy to understand how to plot a simple function, here are several things you should know if you want data plots (which you will):

- You should design your simulation application (i.e. Fortran90 code in our case) so that it outputs data to a simple ASCII text file.
- Write data in columns. Consider, for example, the following datafile (which we will call *falling.data*) representing the time, position, velocity and acceleration of a falling body at several moments of time:

#	Time	Position	Velocity	Acceleration
	10.0	3000.000	-98.077	-8.605
	20.0	2019.232	-184.130	-5.573
	30.0	177.934	-239.857	-2.625

- Assuming we have started *Gnuplot* from within the same directory as the one where our "falling.data" file resides, we can obtain the *position vs. time* plot by typing:

```
gnuplot> plot "falling.data" using 1:2 with lines
```

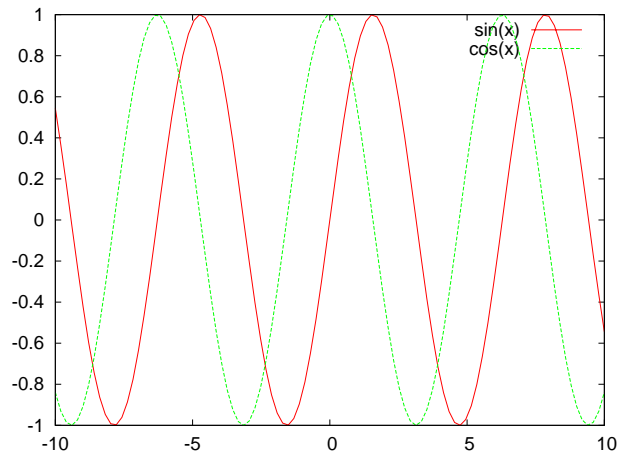


- In the example, *using 1:2* means "using column 1 for X-data and column 2 for Y-data". If you would like a velocity plot instead, you would have to replace this with *using 1:3* (since column 3 represents velocities in our example).
- Another new thing is the "*with lines*" sequence, which specifies the style of the plot (try replacing it with "*with points*" or "*with linespoints*" and see what changes). All styles (*lines*, *points* & *linespoints*) have sub-styles, which one can select by specifying an additional number after the basic style name. For example, try the following:

```
gnuplot> plot "falling.data" using 1:2 with lines 0
gnuplot> plot "falling.data" using 1:2 with lines 1
gnuplot> plot "falling.data" using 1:2 with lines 2
```

- What these numbers actually mean is machine/architecture dependent, so it's not something worth memorizing. The important thing is that they are designed to be easily distinguishable from one another, so that you can do *overplots*.
- **Overplots** can be produced by appending multiple function or datafile specifications (separated by commas) after the *plot* command, as in the examples below:

```
gnuplot> plot sin(x), cos(x)
gnuplot> plot "falling.data" using 1:2, "falling.data" using 1:3
gnuplot> plot "falling.data" using 1:2 with lines, x**2*sin(x)\
with points
```



- **Titles** can be specified using the *title* parameter, as in:

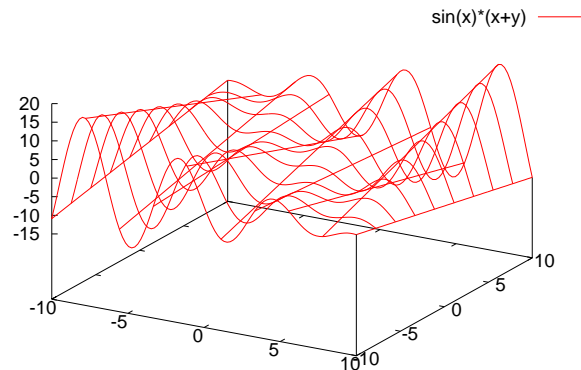
```
gnuplot> plot "falling.data" using 1:2 with lines \
title "Y(t) from datafile", x**2*sin(x) \
with points title "Funny function"
```

(note the backslash `\`, which allows us to split long Gnuplot commands over several lines).

3.2 The *splot* command:

As already mentioned, *splot* is used for surface plots, i.e. functions depending on two variables. Just to get you started, here is a simple example:

```
gnuplot> splot sin(x)*(x+y)
```



Gnuplot will create a *wireframe* depicting your function. Note that you can rotate the view by dragging it with the mouse :). However, the output may be hard to view, because it shows all of the line segments in the wireframe at all points. To fix this, use the following option:

```
gnuplot> set hidden3d
```

Most of the discussion given above for the *plot* command also holds for *splot*, with some modifications in the required datafile format though, which we will now explain.

- For data plots, your simulation application should output data in ASCII format, as in the case of the *plot* command.
- If you want to draw a 2D-field with *splot*, you have two choices for the output format: ***column format*** and ***matrix format***.
 - The ***column format*** allows you to specify the value of the function at arbitrary points in your XY grid. Your data should be grouped in blocks, according to the X-coordinate, as in the example datafile below (which we will call “*test3d.dat*”):

```
# X Y Z
0 0 0
0 1 1
0 2 4
0 3 9
0 4 16
0 5 25

1 0 1
1 1 2
1 2 5
1 3 10
1 4 17

2 0 4
```

```
2 1 5
2 2 8
2 3 13

3 0 9
3 1 10
3 2 13
```

To plot this data, use e.g:

```
gnuplot> splot "test3d.dat" using 1:2:3 with lines
```

Note that the blocks should be separated by blank lines.

In this case, we obtain *3D lines* rather than the *wireframes* from the previous function plots. This is due to the fact that *Gnuplot* only draws surfaces if the datafile contains the values of the function on a complete rectangular grid. This was not true for “*test3d.dat*” above, but is true for “*test3d_reg.dat*” given below:

```
# X Y Z
0 0 0
0 1 1
0 2 4
0 3 9
0 4 16
0 5 25

1 0 1
1 1 2
1 2 5
1 3 10
1 4 17
1 5 26

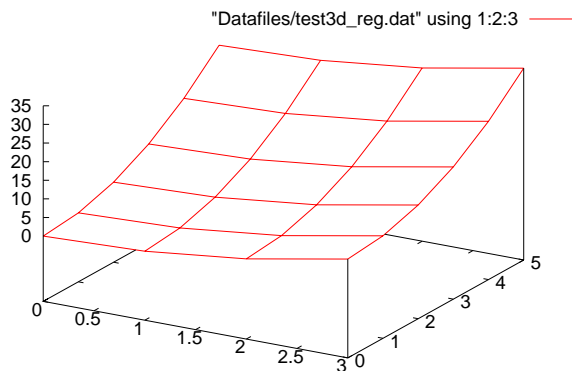
2 0 4
2 1 5
2 2 8
2 3 13
2 4 20
2 5 29

3 0 9
3 1 10
3 2 13
3 3 18
3 4 25
3 5 34
```

If you now plot “*test3d_reg.dat*” using:

```
gnuplot> splot "test3d_reg.dat" using 1:2:3 with lines
```

you should see the nice grid we are expecting.



- The *matrix format* consists of writing in the datafile only the value of the function on a rectangular grid. For example, take the following datafile (which we will call “*test3d_mat.dat*“):

0	1	4	9
1	2	5	10
4	5	8	13
9	10	13	18
16	17	20	25
25	26	29	34

In the *matrix format*, rows are along the X-direction, while columns are along the Y-direction. This datafile can be plotted using the command:

```
gnuplot> splot "test3d_mat.dat" matrix with lines
```

Note that nothing has been said so far about the actual X & Y ranges. *Gnuplot* assumes a $[0 : n] \times [0 : m]$ grid, where n is the # of rows and m is the # of columns. However, one can change that using the *set {x|y|z}tics* command. For example, if we want to map the default X-range of $[0 : 3]$ for the datafile above to $[100 : 300]$, we can issue the following command:

```
gnuplot> set xtics ("100" 0, "200" 1, "300" 2)
```

and then repeat the plotting.

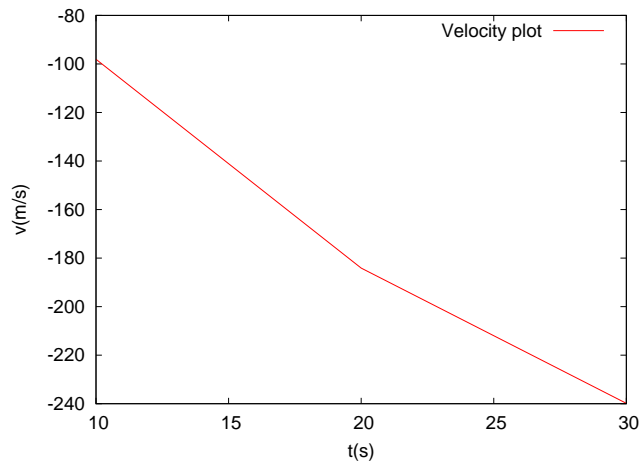
Finally, if the data to be plotted consists of 3D scatter points (as in the case of the integration of the Lorentz system), no special formatting of the input data is necessary (except the arrangement in 3 columns), and they can be plotted using:

```
gnuplot> splot "data_file.dat" using 1:2:3 with lines
```

4 How to do various other useful things:

- **Labelling axes** is done with the *set {x|y|z}label* command. For example, in the case of our 2D plot example, one may use the commands:

```
gnuplot> set xlabel "t(s)"
gnuplot> set ylabel "v(m/s)"
gnuplot> plot "falling.data" using 1:3 title "Velocity plot" with lines
```



- **Range selection** is done with the `set {x|y|z}range [lower bound:higher bound]` command, as in the following examples:

```
gnuplot> set xrange [10:30]
gnuplot> set yrange [0:2000]
gnuplot> plot "falling.data" using 1:2 with lines, x**2*sin(x) \
with points
```

- **Autoscaling** (have Gnuplot determine ranges):

```
gnuplot> set autoscale
gnuplot> plot "falling.data" using 1:2 with lines, x**2*sin(x) \
with points
```

- **Setting the dimensions of your plots (in inches):** is done with the `set size <width>, <height>` command, as in the example:

```
gnuplot> set size 0.5, 0.5
```

- **Linux Shell commands from within Gnuplot:** If you want to execute a single Linux command without exiting *Gnuplot*, you can do this by prefixing the respective command by an exclamation mark (!), as in the example:

```
gnuplot> ! ls -Alh
```

If you need more than one command, use the *shell* command; to revert to the *Gnuplot shell* afterwards, just type *exit*.

- **Plotting only specific points:** You may select to plot only parts of your datafile. This may prove to be especially useful in the cases when the datafiles are rather large. The selection can be done using the `index m:{n}{:p}` command, where *m* is the *starting* index (required), *n* (optional) is the *end* index and *p* (also optional) is the *index step size*. Hence, this translates to "plot datapoints from the one with index *m* to the one with index *n* in steps of *p*". A concrete example of usage is:

```
gnuplot> plot "falling.data" index 0:3:1 with lines
```

Note: alternatively, one may use instead of the *p* variable above the *every* parameter, such as in the example:

```
gnuplot> plot "falling.data" index 0:3 every 1 with lines
```

5 Writing plots into files

By default, Gnuplot will show you the files in a window on the screen. However, you may sometimes want to have the output available as a image file. Of course, there is the hard way to obtain this (gnuplot, screenshot, image processor :)), but it's not a very clever choice. The *right* way to do it is to tell *Gnuplot* to re-direct the output to a *Enhanced PostScript* file. To do this, you will have to type at the *Gnuplot* shell¹:

```
gnuplot> set output "test.ps"
gnuplot> set term postscript enhanced color
```

Note that these commands have to come before the plotting commands to have the desired effect (hence, if you tried the examples so far in *Gnuplot*, you will have to do the plotting again). Every plotting command you issue from now on will be written to the "test.ps" file, each plot on a separate page. When you are done with writing to files, you can restore the *draw-plots-on-screen* policy by issuing the following commands:

```
gnuplot> set output
gnuplot> set terminal x11
```

Tip: We have noticed that the best results are obtained when writing the plots to the *Enhanced PostScript* format instead of plain *PostScript*. This can be done as follows:

```
gnuplot> set output 'test.eps'
gnuplot> set term postscript eps enhanced color
```

6 Gnuplot scripts

Now, all of the above may have seen a little bit of a mess. If you had to type all of the commands every time you wanted to plot something, it wouldn't be much of a gain, right?

This is why *SCRIPTS* were invented :). Basically, a script (*Gnuplot script* in the present context) is a text file which contains the commands you would normally type in the *SHELL*. To give you a starting point, here is a sample script to plot the "falling.data" file and do *EPS* output:

```
# FILE: "falling.p"
# PURPOSE: Plot the dataset "falling.data" and generate .eps image
# NOTES: - As you can see, lines beginning with "#" are considered
#         comments in Gnuplot scripts;
#         - Although not mandatory, we recommend that you give to your
#         plotting scripts the extension ".p" (e.g. "myplot.p")
#
set term postscript eps enhanced
set output "falling.eps"
set size 1.0, 1.0
set autoscale
set xlabel "t(s)"
set ylabel "y(m)"
plot "falling.data" using 1:2 with lines
```

Let us assume you typed all of the above into the file "falling.p", which resides in the same directory as the "falling.data". To use the script, type in the *Linux Shell*²:

¹Of course, replace "test.eps" with whatever you want your picture file to be called.

²Note that the script will not cause anything to be displayed on the screen. To see the output, you need to use a *PostScript* viewer such as **gv**, which should be available on most systems.

```
bash/tcsh> gnuplot falling.p
```

Note that if you typed all of the settings and plotting commands at the *Gnuplot shell* and you are happy with the result, you also have the time-saving possibility of instructing *Gnuplot* to generate your script. This is done as in the example:

```
gnuplot> [some gnuplot instructions here]
gnuplot> [.....]
gnuplot> save "output.p"
```

7 Getting additional help

This document is by no means a comprehensive reference. Fortunately, there are additional resources out there. The most convenient one would be *Gnuplot's* excellent inbuilt help system, which can be accessed directly from the *Gnuplot shell* by typing:

```
gnuplot> help [command]
```

such as in the following examples:

```
gnuplot> help plot datafile
gnuplot> help index
```

Of course, there is also the *WWW*, from which we give several links that we consider most helpful:

```
http://www.duke.edu/~hpgavin/gnuplot.html
http://t16web.lanl.gov/Kawano/gnuplot/index-e.html
```