Institut für Theoretische Biochemie, Universität Wien

Übungen zu Strukturbiologie und Theoretischer Chemie

Sommersemester 2010

Teil 1: Einführung in Linux, Darstellung von 2D/3D Daten und nichtlineare Regression

Ronny Lorenz (ronny@tbi.univie.ac.at)

Contents

1 Make yourself familiar with the LINUX operating system

1.1 Introduction

For the next couple of tasks it will be necessary to use the so called `Terminal` of the Linux operating system. Therefore, you have to get used to some basic commands that make the life easier when navigating through your directories and operating with textual data files.

The `Terminal` can be used to execute commands and start programs. Therefore, the appropriate command has to be typed into the *terminal* window, followed by hitting the Enter button. In the next sections we highlight this by:

```
$ command
```

You can interrupt a program or command that you have started in the *terminal* any time by pressing Ctrl - C . By appending a `&` character at the end of the command, the resulting process will operate in the background, not blocking the *terminal* for further commands. This behavior can also be achieved by pressing Ctrl - Z and entering

```
$ bg
```

while your program is running. An example for starting a program in the background could look like this:

```
$ firefox &
```

If you start writing down a longer path- or a longer command-name, you can use the *auto-completion* feature of the *terminal.* Therefore write down the first letters of your path or command and press the ⇆ key. Doing so, the *terminal* will either complete your started phrase automatically if its unambigous or it will provide you with an overview of available commands or paths that start with what you've already written. You will love this feature, especially when navigating through complex directory structures. ;)

1.2 Useful commands

The started program operates from within your working directory, i.e. the directory where the command was exectued in. The unix directory structure is dissimilar to the directory structure you may know from a Windows operating system. In Linux, as in other Unix like operating systems, the directory structure is organized as a (partially fixed) tree with a single root node *'/'.* All directories and files in the directory tree are internal nodes or leaves of the tree. A path to a certain directory or file is a path along the tree, starting at the root node, where each change in the layer/depth is indicated by the *'/'*-character. Hence, a path like */usr/bin/test* denotes, that starting from the root directory */* there is a directory named *usr* that contains another directory named *bin.* In the latter directory then is a file named *test.*

To find out the current working directory there is the command `pwd` (print working directory). So finding out the current working directory will look similar to this:

```
$ pwd
/home/theochem
```

This directory is somewhat special as it is your so called home-directory. Home directories of users of a Unix based system usually reside in the path */home*. The home directories are then named by the user account, e.g. *theochem* in your cases.

Changing the working directory to a particular path in the directory tree can be done by using the command *cd* (change directory), followed by the path, e.g.:

```
$ cd /usr/bin
```

Omitting the path and simply calling the *cd* command without it results in a change to the users home directory */home/username*.

There are two special, reserved paths that can be used when browsing through the directory structures, *'.'* and *'..'*, where the first denotes the current working directory and the latter its parent. So typing

```
$ cd ..
```

will change the working directory to the parent of the current working directory, i.e. we will navigate to the parental node in the tree.

The command *ls* can be used to show all entries within the current directory.

```
$ ls
```

prints the list of names of all files and directories. To get a detailed view of all files and directories of the current position in the directory tree, you can pass a so called parameter to the *ls* command. Parameters are usually following the command and start with one or two *'-'* letters. E.g.:

```
$ ls -l
```

will tell the *ls* command to create a detailed list view *(parameter -l)*.

To get an overview of all available parameters of a program or command you usually can use the parameter *-h* or *–help*. Try this with the ls command:

```
$ ls --help
```

For a more detailed overview of the functions and parameter options of a command or program it is also useful to look inside the so called **man-pages** or **manual-pages**. This can be done by typing

```
$ man command
```

In order to use the directory structure, we need to be able to create and remove files or directories. It is also very useful to know how to change the name and copy or move a certain file or directory to another location. The command

```
$ touch filename
```

will create an empty text file named *filename*. To get rid of this file (remove it), you can use the command

```
$ rm filename
```

BE AWARE, that this command will really delete your file instead of moving it to some trashbin or something similar as you might expect. **The file will be removed unrecoverable till eternity!** Keep this in mind when using the *rm* command.

To create a new empty directory, you use the command *mkdir* (make directory).

```
$ mkdir butzemann
```

will create a new empty directory named *butzemann.* To remove a directory the command *rm* can be used again. However, the parameter *-r* must be passed to the *rm* command to indicate that you want to *recursively* remove the directory with all its content.

```
$ rm -r butzemann
```

Renaming files and directories is not done with a special *rename* command but with the move-command *mv.* The move command makes it possible to move a certain file or directory to another location. Imagine you have a file named *foo* and a directory named *bar.*

```
$ touch foo
$ mkdir bar
```

To move the file *foo* into the directory *bar*, the move command would look like this:

```
$ mv foo bar
```

This only works, if there is a directory called *bar* in the current working directory. If this is not the case and there is neither a directory *bar* nor a file with this name, the move command will rename the file *foo* to *bar.* So if the second path passed to the move command is nonexistant yet, the first path will be renamed to the second. This works for both, files and directories

```
$ mv foo bar
```

- moves file *foo* into directory *bar* if *foo* is a file and *bar* a directory
- moves directory *foo* into directory *bar* if *foo* and *bar* are both directories
- renames file *foo* to the name *bar*, if *foo* is a file and *bar* is non-existant
- renames directory *foo* to the name *bar* if *foo* is a directory and *bar* is non-existant

Copying files is accomplished by the copy command *cp.* Regular files *source1*, *source2* and so on can be copied to a certain location *destination* in the directory tree by typing

```
$ cp source1 source2 source3 destination/
```

You can use as many source files as you like but make sure, that the target directory (*destination*) exists.

Copying directories to other locations is done by telling the *cp* command to *recursively* copy the given sources. Therefore use the *-r* parameter option:

```
$ cp -r sourcedir1 sourcedir2 destination/
```

As in the previous examples, always use the parameters *–help*, *-h* or *man command* to find out how to deal best with a certain command.

1.3 Working with text files

Printing a textfile into the *terminal* window is done by using the command *cat*. E.g. there is a special file in the directory tree of the linux operating system that tells you about the specifications of the processor(s) in your workstation. This file is located under */proc/* and named *cpuinfo*. So, in order to find out about the CPUs in your computer, you can type

```
$ cat /proc/cpuinfo
```

Sometimes it is more convenient to be able to go through a text file page by page or line by line instead of printing the entire file into the *terminal*. Therefore the little program *less* exists.

```
$ less /proc/cpuinfo
```

This will display all lines of text in the file */proc/cpuinfo* that fits into your current *terminal* window. Using the arrow keys ↑ and ↓ , you can go through your text line by line. The keys Page ↑ and Page ↓ will show the previous or next page of text. Pressing q will quit the *less* program. You can also search inside a text document by pressing \ followed by your query. This will jump to the nearest occurance of the query in your textfile. By pressing n or Shift ⇑ + n you will be directed to the next/previous occurance. As you will deal with a lot of textfiles with data in the next lectures, there are many helper commands/programs that assist you to obtain certain informations. The program *wc* (word count) counts the number of words, lines and characters in the specified text file. The program *grep* will find out if a certain specified string is in your textfile. E.g. if you want to know if you have a pentium cpu in your workstation you might call

```
$ grep Pentium /proc/cpuinfo
```

If so, the programm will print all lines of the textfile where your query was found. It will print nothing, if the query is not found at all. Again, check

```
$ grep --help
```

or

```
$ man grep
```

to find out more about the possibilities you have using the *grep* command.

1.4 Redirecting input and output of a program

In a unix *terminal*, in contrast to the windows command prompt, you are able to redirect all output of any program, e.g. to a textfile. This is done using the so called *piping* feature of the shell. There are three special characters that tell your *terminal* to *pipe* data from a program into a text file (`>`), to *pipe* a text file as input into a program (`<`) or to *pipe* the data printed by one program as input into another program (`|`). E.g. the call

```
$ ls -l > dirlisting.txt
```

will redirect the output of the *ls* program into a textfile named *dirlisting.txt*. Note that using this command any previous content of the textfile *dirlisting.txt* will be overwritten if this file already exists! Otherwise the file will be created. To append the output of a program to an already existing file you can use the special piping symbol `>>`

```
$ ls -l >> dirlisting.txt
```

These features are very powerful if you want to use several commands to process your data where the data output printed in each intermediate step is only used as input for the next steps and not needed after processing all commands. E.g.

```
$ ps -e | grep -i terminal | wc -l
```

will first use the output of the *cat* program for the file */proc/cpuinfo* as input for the grep program, which in turn searches for lines containing the string "terminal", ignoring the characters case (case insensitive, -i). As we already know, the *grep* command prints out the lines where it finds the search phrase. This output will be used as input for the *wc* program which then counts the number of lines in its input. This is useful for example if you want to know how many times the *terminal* program runs on your system.

Another way to do the same would be calling each program seperately and piping its output into a new textfile. Then, using the textfile produced by each previous step as input, the next command is invoked and again, the output has to be written into a new textfile.

```
$ ps -e > textfile1
$ grep -i < textfile1 > textfile2
$ wc -l < textfile2
```

The problem doing it this way is that we now have two textfiles that we have to delete as we do not need the residing data anymore. Using the example with the `|` piping above we circumvent this problem by not generating intermediate textfiles in the first place. So if you get more involved with the *terminal* it might be very useful to use piping.

1.5 Textfiles with tabulated data

The data files we will deal with after the introduction will always be textfiles with data seperated by spaces or another special character like 'comma'. Each line of the data files will contain a complete data set. For example have a look

into the file
`http://www.tbi.univie.ac.at/~ronny/Leere/sb1/urbanareas.tsv`
that contains the populations of large urban areas arround the world at different points in history.

In order to only show certain columns of data or swap columns, e.g. only the names of cities listed in the data file above, you can use the *awk* program. Here the program call looks slightly more complicated than the ones you have previously seen but the general usage in our cases will mostly stay the same. Use the command

```
awk '{print $1}' urbanareas.tsv
```

to show the first columns of data, i.e. the city names. Use

```
$ awk '{print $1,$3}' urbanareas.tsv
```

to print the citynames followed by the country they are located in. You see, the $x'es after the *print* denote the number of the column.

```
$ awk '{print NF}' urbanareas.tsv
```

will print the number of columns (Number of Fields) in your datafile.

Find out the number of the column that contains the population of the cities in year 1980 and print a list of city names, followed by their population at this time, the country and the geographic postition (altitude/longitude).

We might have touched only the tip of the iceberg with such rather simple executions of the *awk* program. Indeed, it is much more powerful but would exceed the time available to go further into detail. (check the man-pages of awk for more information if you like)

Nevertheless, the last example usage of *awk* will show, how to print only a certain range of datasets from our datafile. First find out the number of lines in the data file using the *wc* command

```
$ wc -l datafile
```

Than, print the city name followed by the population in the year 1950 for the last twenty cities in our datafile

```
$ awk 'BEGIN{i=1}{if(i>=x) print $1,$2; i++}' urbanareas.tsv
```

where x has to be replaced by the line number of the beginning of the block with the final 20 cities. In detail, the statement above tells *awk* to set a variable i to the value 1 in the beginning. Then for each row of processed data this variable is incremented (`i++`). Additionally, *awk* will only print the data column 1 and 2 if the variable i is greater or equal to the specified value x. This behavior can also be achieved using the special variable NR which represents the current row number. So a call of *awk* that does the same would look like this:

```
$ awk 'NR>=x {print $1,$2}' urbanareas.tsv
```

As you might guess, the *awk* command is very useful for slicing out blocks of data from a data file for further usage.

Two other very useful programs are *head* and *tail*. With them, you are able to print the first n lines (*head*) or the last n lines (*tail*) of a file. You can invoke these programs like this:

```
$ head -n 20 datafile and
$ tail -n 20 datafile
```

For further options these programs may provide check the manual pages or the help parameter again.

This should be enough for a first very basic introduction about how to deal with text files in the Linux OS. You should now be able to do a lot of 'magic' stuff with the data you will be provided with. ;)

2 2D plots

In this section we are going to plot several data. First create a new directory in your home that will contain the data files we are going to use. Open a *terminal* and change into your home directory, if you are not already there. Create the new directory and name it as you wish, e.g.:

```
$ cd
$ mkdir data_directory
```

Change into that newly created directory. Open a webbrowser and download the data files we are going to use to the data directory you've just created. Alternatively you can stay in the *terminal* and use the program *wget* that makes it possible to download webcontent to your working directory

```
$ wget URL
```

The data we will use is available under the following URLs:
`http://www.tbi.univie.ac.at/~ronny/Leere/sb1/data.1`
`http://www.tbi.univie.ac.at/~ronny/Leere/sb1/data.2`
`http://www.tbi.univie.ac.at/~ronny/Leere/sb1/data.3`
`http://www.tbi.univie.ac.at/~ronny/Leere/sb1/data.4`
`http://www.tbi.univie.ac.at/~ronny/Leere/sb1/data.5`

After downloading all data, we are prepared to use the programs *grace* and *gnuplot* to plot the provided data.

Before plotting anything with one of the following programs, take some time to have a look inside the data files to ensure that the residing data fits your expectations.

Hint: Use

```
$ head -n 20 datafile
```

to see the first 20 lines...

2.1 Grace

Lets first focus on Grace:

Invoke *grace* using the first data file by using the command:

```
$ xmgrace data.1
```

This will open a new window with an x-y plot of the data in *data.1* Do the same with all other data files and play arround with some settings in the Grace window to adjust the axes, coloring etc...

Just make yourself a bit familiar with all the settings possible.

2.2 gnuplot

Gnuplot is another program to visualize data. But, in contrast to *grace*, *gnuplot* does not provide you a window with menues to change viewing options.

```
$ gnuplot
```

will start an interactive *gnuplot* session that is somehow similar to the usage of a *terminal*.

You can easily plot the data in the first file by typing:

```
gnuplot> plot "data.1"
```

This will produce an x-y plot of the data in file *data.1*, where *gnuplot* adjusts the x- and y-axis according to the data provided. By default, the data points from the data file are plotted as little plus sings `+`. You can change this behavior by appending *with lines* to the plot command:

```
gnuplot> plot "data.1" with lines
```

Additionally, *gnuplot* assumes the data to be provided as one or two columns in your data file. If there is just one column in your file, *gnuplot* uses that data for the y-axis. Otherwise, if two columns of data are provided, the first column denotes the x- and the second the y-coordinate. If your data file has more than two columns or your data is provided in another order, you can tell *gnuplot* which of them should be used for which coordinates with the statement *using*.

```
gnuplot> plot "urbanareas.tsv" using 4:5 with lines
```

This will produce a plot of the data in *urbanareas.tsv* where the 4th column is used as x- and the 5th column as y-coordinate.

In order to adjust the plot to your own needs, e.g. the range of the x- or y-axis, colors, logarithmic scales and so on, there exist several settings that can be set/unset using the commands *set* and *unset*

```
set xrange [0:1000] will set the range of the x-axis to the
                    interval [0:1000]
set yrange [-5:10]  similar to xrange
set logscale x      turns on logarithmic scaling for the x axis
set logscale y                    - "" -               y axis
unset xrange        unsets a specified xrange, i.e. the range
                    of the x-axis will be determined by the data
unset logscale x    unsets logarithmic scaling for the x axis
```

Consult the implemented *help* function within the *gnuplot* interactive shell by typing

```
gnuplot> help
```

or

```
gnuplot> help plotting
```

Alternatively, you can visit the website
`http://t16web.lanl.gov/Kawano/gnuplot/index-e.html`
to obtain more information about how to tweak the output. Using this website, find out how to save your graph as a postscript file that you may include in any scientific work.

You can leave the interactive *gnuplot* session by typing

```
gnuplot> exit
```

3 Curve fitting / Nonlinear regression

After inspecting the data given in the files you just downloaded with *grace* and *gnuplot* we return to *grace* for the next task of fitting a function to the data provided.

Open the menu entry `Data/Transformations/Non-linear curve fitting`. Select your dataset in the left half of the window (Source). Enter a polynome of the variable x with parameters $A_0, \ldots, A_N$ in the `Formula` field of the `Main` tab. Type in a non-linear function (formula) like this:

```
y = A0 + A1 * cos(A2 * x)
```

Then select the number of variables you've introduced in the drop-down menue and provide good estimates for these variables if possible.

You can use as many parameters as you want and make the functions as complicated as possible. Mathematical operators you may use inside you non-linear polynome are:

```
+: addition,       e.g.: (a + b)
*: multiplication, e.g.: (a*b)
^: power of,       e.g.: (x^2)
```

It is also possible to use trigonometric functions like `sin(), cos()`, the logarithm `log()` or the exponential `exp()`

Be aware of the fact that you should first have a look at the data curve itself to get a good guess of the underlying function. If you can't find any function that fits your data or if the data does look too strange try other ways to find out properties of the underlying data. *Grace* provides you with tools for analyzing the distribution of the values (you can make histograms, Fourier transformations, etc.). You can find the appropriate menu entries under `Data/Transformations`. After examining certain properties of the distribution of your data you might be able to get an idea of the underlying function that produced the data

4 3D plots

As you might have found out, one of the data sets does not provide 2- but 3-dimensional data. Unfortunately, *grace* is not able to plot 3D so we have to switch back to *gnuplot* again...

Start *gnuplot* and use the *splot* command to plot the data into a 3D coordinate system

```
gnuplot> splot "datafile"
```

Like in the previous 2D plot, *gnuplot* uses + signs to mark your data points. Change that behavior by appending *with lines* to the *splot* command again. This will draw a line that connects all data points in the order the data was given. Of course, this might produce ugly looking results, so *gnuplot* provides you with a way to fit your data points into a grid of given precision. After that procedure you will be able to plot the resulting grid instead of just the data itself.

This is especially useful if you do not have data that is distributed in a grid like way but scattered. Use

```
gnuplot> set dgrid3d x,y,w
```

and replace x with the number of gridpoints in x direction, y with the number of gridpoints in y direction and w with a weighting factor $(1, 2, 4, 8)$. The weighting factor w influences the degree of weighting of the given z value. Each z value is weighted inversely by the distance from the gridpoints raised to the power of the weighting factor (norm). The parameter w may be omitted. Default values are $x = y = 10$ and $w = 1$. See also

```
gnuplot> help dgrid3d
```

After setting a reasonable grid, type

```
gnuplot> replot
```

to plot the data last used again.

Try different grid sizes and weighting norms to investigate their influence on the data.

4.1 2D projections of 3D data

In most cases of 3D plot you can rotate the coordinate system as much as you want in any direction but never get a good overview of all data provided.

This problem can be circumvented by depicting the 3rd dimension by something else than an additional spatial dimension in the underlying coordinate system. A possibility to do so is for example color- or intensity-coding using gradients and/or step functions.

You can do this easily in *gnuplot* by setting the *pm3d* option (palette mapped 3D). NOTE that this feature can only be used when our data is converted into a grid as we did before with *dgrid3d*

First consult the help provided with

```
gnuplot> help pm3d
```

Then make a first color coded plot by typing

```
gnuplot> set pm3d
gnuplot> replot
```

Deactivating the color encoded third dimension again is done by entering

```
gnuplot> unset pm3d
```

It is also posssible to make a map of the color encoded 3D data at the bottom of the graph with

```
gnuplot> set pm3d at b
```

To hide the lines of the graph and only show the *pm3d* generated color encoding, you can type

```
gnuplot> set hidden3d
gnuplot> replot
```

For additional features visit the following webpage again:
`http://t16web.lanl.gov/Kawano/gnuplot/index-e.html`

4.2 Gri - a programming language for drawing science-style graphs

A more complex example of producing 2D projections from 3D data will be highlighted in this section. Here, we use a program similar to *gnuplot*, named *gri*. It also provides you with an interactive *terminal* where you can type in several commands for data source selection, data post-processing and alteration of the resulting plot.

You can download the *gri*-reference- and the *gri*-command-reference-card under
`http://gri.sourceforge.net/refcard.pdf` and
`http://gri.sourceforge.net/cmdrefcard.pdf`

As you see in the reference cards, it is a very powerful language for drawing science-style graphs. We will rely on a small example that should just show you what may be possible if you have 3D data and want to produce nice graphs to include in your diploma- or Phd-thesis.

Usually *gri* does not run interactively as it produces a postscript image of the resulting graph that can not be viewed in the intermediate steps. Therefore, all commands you may want to pass to *gri* can be written into a textfile *commands.gri* that has the following structure:

```
command 1
command 2
command 3

...

command n

quit
```

Gri will execute all the commands listed in your commandfile *commands.gri* when you invoke it in the following way:

```
$ gri commands.gri yourdata.dat
```

You will be provided with an example command file and some example data. Download it here:
`http://www.tbi.univie.ac.at/~ronny/Leere/sb1/commands.gri`
`http://www.tbi.univie.ac.at/~ronny/Leere/sb1/sv11.2D.out`

```
http://www.tbi.univie.ac.at/~ronny/Leere/sb1/switch.2D.out
```

E.g. using *wget*:

```
$ wget http://www.tbi.univie.ac.at/~ronny/Leere/sb1/commands.gri
$ wget http://www.tbi.univie.ac.at/~ronny/Leere/sb1/sv11.2D.out
$ wget http://www.tbi.univie.ac.at/~ronny/Leere/sb1/switch.2D.out
```

First have a look inside the first data file.

```
$ less sv11.2D.out
```

It is filled with a lot of information, not entirely given as tabulated columns. In short, this data represents the output of a bioinformatics tool that computes the free energy of a given RNA molecule. RNA can can form stable secondary structures by base pairing and base stacking. A widely used nearest neighbor model with tons of parameters is then used to compute the free energy of a given secondary structure of a particular RNA. Furthermore, there exist algorithms that compute the minimum of free energy and the according secondary structure. RNA does not reside in a fixed configuration but may change its secondary structure. Thus, another algorithm computes the partition function of all secondary structure conformations compatible with the RNA sequence. Given the partition function one can calculate the Gibbs free energy of the ensemble of secondary structures. If a neighborhood relation is introduced that tells which secondary structure is a neighbor of another, e.g. if you delete/introduce a base pair, you get a so called *energy landscape*. This requires that each secondary structure state is associated with a certain free energy value. This energy landscape is a high dimensional raum (the number of dimensions relies on the number of neighbors a secondary structure may have) that can not be easily depicted. However, given one or two reference structures, one is able to project the entire high dimensional space onto a 2- or 3-dimensional plane. In our example this is done by partitioning all secondary structure states into so called *distance classes* originated by two given reference structures. This means that for each possible secondary structure state, the distance to the two reference structures according to the underlying neighborhood relation is computed which results in a (x,y) pair where x is the distance to the first and y the distance to the second reference structure. This partitioning can than be used to compute the representative with a minimum of free energy and/or the partition function/Gibbs free energy of the partitions. However, this is the data inside the example given.

In the first line of *sv11.2D.out* is the sequence of the RNA molecule. The second line denotes a secondary structure with minimum of free energy in dot-bracket notation, followd by the fre energy in kcal/mol. The third and fourth line show the two reference structures with their appropriate free energy. In the next line you see the Gibbs free energy of the ensemble of all secondary structures compatible with the sequence given. Then a headerline of the following tabulated data is given that tells you which data is given in which column. The first column (k) is the distance to the first reference, the second (l) the distance to the second reference. Then there is a set of probabilities followed by the minimum free energy, the gibbs free energy and the minimum free energy structure of the partition.

We will use this data to draw a 2D projection of the following 3D data: (distance1, distance2, MFE)

Therefore consult the given command file *commands.gri*. **Do not get confused with all the statements written there!** Before each statement there is a line that starts with `//`. This is recognized as a comment and does not influence the plot produced by *gri*. It just helps you to know what is done in the next step.

To understand what is going on when *gri* is executing all the commands given in *commands.gri*, execute the command file using the provided data first.

```
$ gri commands.gri sv11.2D.out
```

The command file you got has a very special line on top:

```
#!/usr/bin/gri
```

This makes it possible to execute the textfile itself in the *terminal*, without writing the command *gri* in front. To do so, you first have to make the textfile executable using the command *chmod*:

```
$ chmod +x commands.gri
```

Now you are able to produce the postscript plot using the command file alone:

```
$ ./commands.gri sv11.2D.out
```

If you get an error message in your *terminal* that contains a line like:

```
bad interpreter: No such file or directory
```

you have to find out the exact path where the *gri* preogram is located in your directory tree. This can be done typing

```
$ which gri
```

or

```
$ whereis gri
```

You should now get a path like */usr/bin/gri* or something similar if *gri* is installed in your system. Copy this path and replace the part after `#!` in the first line of your *commands.gri* with it. This line, by the way, tells the *terminal* which program it should use when executing a text file.

After everything went fine, check the content of you current working directory with the *ls* command

```
$ ls
```

You should now see a file named *sv11.2D.out.ps* that contains the plot produced by *gri*. This file can be viewed by any program that can read the postscript format, e.g. *gv, evince, etc.*.

```
$ evince sv11.2D.out.ps
```

You can also convert the postscript image to a *PDF* using the *ps2pdf* command:

```
$ ps2pdf sv11.2D.out.ps
```

This produces a file named *sv11.2D.out.pdf* that can be viewed by any *PDF*-viewer.

After looking at hte produced plot, go through *commands.gri* to find out which commandss were necessary to produce the plot.

You can easily comment out some commands in the command file by putting a `//` in front of it...

Online resources that might be helpful:

- *Grace* related
  - Grace website
    `http://plasma-gate.weizmann.ac.il/Grace/`
- *Gnuplot* related
  - Gnuplot website
    `http://www.gnuplot.info/`
  - Einführung in gnuplot - Rechenzentrum Uni Osnabrück
    `http://www.rz.uni-osnabrueck.de/Zum_Nachlesen/Skripte_Tutorials/Gnuplot_Einfuehrung/pdf/gnuplot.pdf`
  - Gnuplot reference card
    `http://www.gnuplot.info/docs_4.0/gpcard.pdf`
  - Not so frequently asked questions about gnuplot
    `http://t16web.lanl.gov/Kawano/gnuplot/index-e.html`
- *Gri* related
  - Gri website
    `http://gri.sourceforge.net`
  - Gri reference card
    `http://gri.sourceforge.net/refcard.pdf`
  - Gri command reference card
    `http://gri.sourceforge.net/cmdrefcard.pdf`
  - Gri documentation
    `http://gri.sourceforge.net/gri.pdf`
- unrelated
  `http://www.google.com` ;)

Good luck and may this tutorial help you for accomplishing further tasks!