

Workshop-Script:
Scientific Figures with GMT4 β
& Bash-Programming

Malte Thoma*

October 1, 2012

!!! This is a draft !!!

**Please be aware that it contains several errors.
However, feel free to use it to learn something about
GMT and bash-programming.**

Please send comments and corrections to Malte.Thoma@awi.de

I hear and I forget.
I see and I remember.
I do and I understand.
(Confucius)

*Many thanks to Jörg Robl for his contributions to sections 1.1 and 7.1, Christoph Oelke for his contribution to section 5.4, Florian Wobbe and Vikram Unnithan for their additional ideas for section 5 as well as Tobias Linke for the translation of this script.

Contents

1	Introduction	5
1.1	„What is GMT?“	5
1.2	Official documentation	5
1.3	Prerequisites for this workshop	6
1.4	Working Environment	6
2	Simple maps	7
2.1	Command line	7
2.2	Shell-Script	7
2.3	Explanations to the use of <code>pscoast</code>	8
2.4	Exercise 1: <code>pscoast</code>	8
3	X–Y–plots with the command <code>psxy</code>	9
3.1	Usage of standard input	9
3.2	Exercise 2: <code>psxy</code> , axis labels, color	10
3.3	Reading input from a file	11
3.4	Exercise 3: <code>psxy</code> , <code>pstext</code> , special characters	12
4	Linux-tools & bash-programming	13
4.1	Exercise 4: <code>psxy</code> , <code>grep</code> , <code>awk</code> , logarithmic projection	14
4.2	Some hints for <code>bash</code> programming	16
4.3	Functions and queries in the <code>bash</code>	16
4.4	Command line arguments for <code>bash</code> programs	18
4.5	Exercise 5: Command line arguments for <code>bash</code> programs	19
4.6	Incidental remark	20
4.7	Additional exercise 6: <code>bash</code> -programming	20
5	Maps, cities, legends and more	23
5.1	Exercise 7: Extern data, transparency, <code>pslegend</code>	23
5.2	Exercise 8: Something about projections	25
5.3	Exercise 9: Plotting ship track GPS-data on a map	26
5.4	<code>psxy</code> and the date format	26
6	Representation of data with two independant variables	29
6.1	Simple 3D graphs with <code>psxyz</code>	29
6.2	Exercise 10: <code>psxyz</code>	29
6.3	2D graphs with <code>grdcontour</code>	30
6.4	Exercise 11: <code>grdcontour</code>	30
6.5	2D plots with <code>grdimage</code>	31
6.6	Exercise 12: <code>grdimage</code>	31
6.7	Exercise 13: 3D graphs with <code>grdview</code>	32
7	Creation of grid files in the netCDF-format	34
7.1	Digital height models	35
7.2	Exercise 14: DEMs and <code>xyz2grd</code>	36
7.3	Gridding of data	37

A Colormaps	40
B Useful tools	42
B.1 Distance of two points on the earth surface	42
B.2 Tangent	42
B.3 Correlation coefficient	43
C Sample Solutions	44
C.1 Solution to Exercise 5	44
C.2 Solution to Exercise 6	45
C.3 Solution to Exercise 7	47
C.4 Solution to Exercise 8	48
C.5 Solution to Exercise 9	50
C.6 Solution to Exercise 12	51
C.7 Solution to Exercise 14	53

Bash–scripts, examples and solution

2.1 First example: <code>pscoast</code>	7
3.1 <code>psxy</code> , usage of standard input	9
3.2 <code>psxy</code> , <code>gmtmath</code> , input from a file	11
4.1 <code>psxy</code> , <code>grep</code> and <code>awk</code>	14
4.2 <code>psxy</code> , <code>minmax</code> , bash programming, <code>grep</code> and <code>awk</code>	17
4.3 bash-program to write a header or a footer for GMT figures.	18
5.1 <code>psxy</code> with date format	27
6.1 <code>psxyz</code>	29
6.2 <code>grdcontour</code>	30
B.1 Calculation of the distance (in km) between two points with given geographical coordinates.	42
B.2 Calculation of a tangent for a (x, y) dataset.	42
B.3 Calculation of the correlation coefficients (and the variances) of a (x, y) Dataset.	43
C.1 Exercise’s solution 5 (<code>write_head_foot.sh</code>)	44
C.2 Exercise’s solution 6 (<code>bash_task.sh</code> , Part 1)	45
C.3 Exercise’s solution 6 (<code>bash_task.sh</code> , Part 2)	46
C.4 Exercise’s solution 7 (<code>land_coloured.sh</code>)	47
C.5 Exercise’s solution 8 (<code>projections_task.sh</code> , Part 1)	48
C.6 Exercise’s solution 8 (<code>projections_task.sh</code> , Part 2)	49
C.7 Exercise’s solution 9 (<code>nmea.sh</code>)	50
C.8 Exercise’s solution 12 (<code>grdcontour_task.sh</code> , Part 1)	51
C.9 Exercise’s solution 12 (<code>grdcontour_task.sh</code> , Part 2)	52
C.10 Exercise’s solution 14 (<code>dem.sh</code> , Part 1)	53
C.11 Exercise’s solution 14 (<code>dem.sh</code> , Part 2)	54

List of Figures

3.1 Example for <code>psxy</code>	10
3.2 Example for <code>psxy</code> und <code>pstext</code>	13
4.3 Example for a logarithmic projection and <code>gmtset</code>	14

4.4	Plot for Exercise 6.	21
5.5	Examples for creative use of <code>pscoast</code> , <code>psxy</code> and <code>pslegend</code>	24
5.6	Projections and node connections on a sphere I	25
5.7	Projections and node connections on a sphere II	25
5.8	Plotting a ship track from GPS-data.	26
5.9	Examples for date and time scales	28
6.10	Example for <code>psxyz</code> and Exercise 10.	33
6.11	Examples for <code>grdcontour</code> and Exercise 11.	33
6.12	Topography of Europe and the Alps with <code>grdimage</code>	33
6.13	Topography of Europe and the Alps with <code>grdview</code>	34
7.14	Graphic representation of the SRTM30-data (several examples).	38
A.15	The RGB color scheme	40
A.16	The GMT CPT colormaps	41

1 Introduction

1.1 „What is GMT?“

GMT stands for *Generic Mapping Tools*. It is a software package that can be used for processing and graphical representation of data. The GMT developers summarise it as follows:

GMT is a free, open source collection of ≈ 60 UNIX tools that allow users to manipulate (x,y) and (x,y,z) datasets (including filtering, trend fitting, gridding, projecting, etc.) and produce Encapsulated PostScript File (EPS) illustrations ranging from simple x - y plots through contour maps to artificially illuminated surfaces and 3-D perspective views in black and white, gray tone, hachure patterns, and 24-bit color. GMT supports 25 common map projections plus linear, log, and power scaling, and comes with support data such as coastlines, rivers, and political boundaries.

GMT was and still is being developed by Paul Wessel and Walter Smith. The software is licensed under the GNU-license. GMT is written in ANSI C standard (Kernighan & Richi 1988) and therefore runs on nearly every system where a C-compiler is available. It runs under Windows, Unix, Linux, MacOS, BEOS and other operating systems, but full performance can only be achieved in combination with shell-programming. For this reason the combination of Unix/linux and GMT has been established. If someone wants to stick to Windows, the Linux emulation CYGWIN is an option.

GMT can be handled via command lines (similar to DOS) or, more efficiently, using shell scripts. There is no GUI (Graphical User Interface) with menu control or buttons to click on. This may appear to be a disadvantage at first, but working intensively with GMT this proves to be its actual strength.

Most Windows applications more and more become “Swiss Army knives“ (and hence need more and more resources). In contrast, GMT chose to use UNIX. Each task is carried out by a small and flexible program. This modular concept makes it possible to incorporate – via shell programming and Unix/Linux – tools such as `awk`, `cat`, `grep` etc. into GMT-shell-scripts. The advantages are obvious:

1. Only the programs needed are loaded into the memory.
2. Each of those programs is tiny in comparison to proprietary software as ArcView, CorelDraw, or Word.
3. Each individual operation is independent. Hence, errors can be localised easily.
4. The individual tools can be combined in shell scripts, data can be transferred via pipes.

Maps and graphics in many well-known journals such as JGR, EOS, EPSL and others are in large parts computed with GMT. This is on the one hand due to its special flexibility through script programming, on the other hand due to the *aesthetic value* of the maps.

1.2 Official documentation

The official GMT documentation can be consulted online:

http://gmt.soest.hawaii.edu/gmt/html/gmt_services.html

In particular, there is a number of helpful supplements that are beyond the scope of this workshop. This includes, amongst others:

- Compilation of all GMT commands.
- Explanation of all possible projections with a graphical example.
- A useful ‘cook-book’ which includes numerous examples for complex graphics.
- The technical reference with tables for patterns, octal codes for special characters, fonts and range of colors.

1.3 Prerequisites for this workshop

- A linux account (username and password) is essential.
- Knowledge of the functionality of a shell (or `xterm`) where commands can be prompted.
- Knowledge of the most important UNIX-commands. These are (without claiming to be complete): `mkdir`, `ls`, `cd`, `cp`, `mv`, `rm`, `man`, `chmod`, `ssh`, `scp`. Please make yourself familiar with them, if you do not know them. You can do this for instance by consulting the online manuals (e.g. `man ls` or `man man`). It is not necessary to know all options of a command. It is sufficient to know what the command generally does and how to obtain more information about the options (with `man`).
- Knowledge about the functionality of the `<TAB>`-key and the `↑`-key in the shell (or `xterm`).
- Knowledge about the X-Windows clipboard (cut & paste).
- A computer with a complete GMT installation needs to be accessible. If this is not the case on a local machine, you have to use `ssh` to log in to the respective machine (possibly with an explicit X- redirection), e.g. `ssh -X limbig.dmawi.de`.
- You need a text editor and know how to use it. Examples for text editors are `vi`, `vim`, `jed`, `joe`, `emacs`. An example for a graphical text editor would be `gedit`. It is important that you feel familiar with *your* editor. If you do not have sufficient experience with any one, I recommend `joe` – it is small, fast, configurable, potent and has a good help function. Online help and tutorial can be found here:

http://heather.cs.ucdavis.edu/matloff/public_html/Joe/NotesJoe.NM.html

The following tasks must be performed with the text editor:

- open a file
- save a file
- close the editor
- mark, copy and move words, lines, paragraphs
- find & replace (if possible with 'placeholders' or so called *regular expressions*)

1.4 Working Environment

- Create a working directory (e.g.: `mkdir GMT`) and move into this directory (`cd GMT`)
- Execute the following commands


```
mtn -d mtn.db db init
mtn -d mtn.db pull apps3.awi.de de.awi.GMTCourse
mtn -d mtn.db co -b de.awi.GMTCourse
```

 to set up your monotone-database, pull the GMT-example scripts from the server, and to checkout the `de.awi.GMTCourse` branch.
- To get updates later, you might have to use


```
mtn -d mtn.db pull apps3.awi.de de.awi.GMTCourse
```

 (`mtn pull` from within your `de.awi.GMTCourse` directory might be enough, if the defaults are set accordingly.)


```
mtn update
```

 (from within your `de.awi.GMTCourse` directory)
- Create your working directory you want to run the course in, e.g. `mkdir course`
- Execute `cp de.awi.GMTCourse/gmtdefaults4.base course/`

2 Simple maps

2.1 Command line

Entering the command

```
pscoast -JN0/15 -R-180/180/-90/90 -Bg30/g15 -G150 -A10000
```

shows the PostScript-code created by `pscoast` in the shell (the meaning of the individual options will be explained in 2.3). In general that does not make much sense; it is more useful to redirect the output into a file with the extension `.ps` (hint: use the `↑`-key):

```
pscoast -JN0/15 -R-180/180/-90/90 -Bg30/g15 -G150 -A10000 > bsp.ps
```

The file `bsp.ps` now contains the figure created by `pscoast` and can be viewed, e.g. with the program `gv`: `gv bsp.ps`.

2.2 Shell-Script

If a command is used regularly (with slightly changed options), it is complicated to enter it manually every time. To avoid that typing work, one can use a *bash-script* to execute several commands in a row. For our examples the script would look like that:

```
pscoast -JN0/15 -R-180/180/-90/90 -Bg30/g15 -G150 -A10000 > bsp.ps
gv bsp.ps
```

It would work, but it has some (more or less) obvious disadvantages. A better example is Script 2.1. In the next section the recurrent elements of *bash*-programming are explained by looking at that script.

Script 2.1 First example: `pscoast`

```
1 #!/bin/bash
2 cp gmtdefaults4.base .gmtdefaults4
3 OUT=pscoast.ps
4 PRO=-JN0/15
5 REG=-R-180/180/-90/90
6 ANN=-Bg30/g15
7
8 # This is a comment
9 pscoast $PRO $REG $ANN -G150 -W1 -A10000 > $OUT
10
11 gv $OUT
12 rm $OUT # Another comment
```

- The first line forces the execution of the script as a *bash*-script (independent from the shell actually used). That line should be found in *every* script.
- Line two ensures a consistent default `.gmtdefaults4` at the beginning of our script.
- In the lines three to six the variables `OUT`, `PRO`, `REG` and `ANN` are assigned (attention: no space before and behind the `=`).
- Comments in *bash*-scripts begin with a `#`.
- The lines nine, eleven and twelve contain the actual commands.
- Options are passed to a program (in this case `pscoast`) with a leading `-`.
- A `$` (as found in the lines nine, eleven and twelve) in combination with a variable returns its value.
- Line twelve deletes the created file.
- Blank lines are for better readability.

In general, there are two reasons to save options (or something else) in variables (and not enter them directly):

1. Very long options (e.g. `-B`) make it hard to understand the script and
2. options that are used more than once had to be adjusted in all locations in the script if the value of the option changes. (In Script 2.1 this is only relevant for the output-file `pscoast.ps` (`OUT`), but it is still common sense to use variables for regularly used options as `-J` and `-R`.)

Options that are just relevant for one command (in this example it is `-G`, `-W` und `-A`) are written directly behind the command.

2.3 Explanations to the use of `pscoast`

The options for the command `pscoast` used in Script 2.1 shall be explained:

- `-J` defines the kind of projection (in the example `N` is chosen, a 'Robinson'-projection). More information about the different kinds of projection can be found in Section 8, the GMT-manual http://gmt.soest.hawaii.edu/gmt4/gmt_services.html or simply (but without examples) with the command `man psbasemap`.
- `-R` sets the plotted region (in this case the whole earth).
- `-B` sets the labels of the axes (here none), the tic-interval (here none) and the size of the grid (here 30° in X-direction (longitude) and 15° in Y-direction (latitude)).
- `-G` defines the color of dry land (0=black, 255=white).
- `-W` sets the line width for the boarder of the continents.
- `-A` sets the minimum size of structures (in km^2) that are shown in the map. Anything smaller will not appear in the plot.

2.4 Exercise 1: `pscoast`

1. Open three `xterms` and place them on the display so you can work with all of them (alternatively you can replace one `xterm` with a graphical editor like `gedit`).
2. Move into your working directory for this course, e.g. with

```
cd ~/GMT/course
```

 (this holds for all examples in this course).
3. Copy the Script 2.1 with the command

```
cp ~/de.awi.GMTCourse/in_pscoast.sh .
```

 and open it with your favourite editor.
4. Check if the script is executable from within a `xterm`. (`ls -l` should show an `x` for the user access permissions. If it is not executable use the command `chmod u+x in_pscoast.sh` to change the access permission (`u`=user, `+`=add, `x`=executability).)
5. Execute the bash-script, by entering the command `./in_pscoast.sh`. As result a plot should appear on the display.
6. Open the manpage for `pscoast` in another `xterm` (`man pscoast`).
7. Experiment with the different parameters of the option and read the corresponding section in the manpage:
 - (a) Test other gray tones `-G` (Colors are subject to Exercise 2).
 - (b) Test which changes can be achieved by using the option `-I` (with different values).
 - (c) Test which changes can be achieved by using the option `-N` (with different values).
 - (d) Try other values for the region `-R`.
 - (e) Test other projections (e.g. `-JW0/15`, `-JQ180/15`). Attention: Not every projection is capable of displaying the whole earth from pole to pole. An example for that is the mercator-projection (next point):
 - (f) Test the projection option `-JM15`. Adjust the region option `-R` until you don't have any error messages. Try different regions.

- (g) Test e.g. `-JS10/90/15 -R-30/50/35/72` and `-JS0/90/15 -R-30/50/35/72`.
What is the difference?
- (h) Try to create a title and axes labels with option `-B`, man `psbasemap` might be helpful. If you can not solve this task, wait for Script 3.1 where option `-B` will be explained in all details.

3 X–Y–plots with the command `psxy`

The most common task is the graphical representation of a function $y = f(x)$. Therefore, the command `psxy` is used. There are two possibilities to submit data to `psxy` (and most other GMT-commands), one can use an ASCII-file or the standard input. Both possibilities are introduced in this chapter. (For the sake of completeness it is to be mentioned that all GMT-commands are also able to read binary data (instead of ASCII), but this is rarely practically relevant.)

3.1 Usage of standard input

In this section it is explained how data is read via standard input. Furthermore some of the many plot options offered by `psxy` are explained. Starting point for this section is Script 3.1.

Script 3.1 `psxy`, usage of standard input

```

1  #!/bin/bash
2  cp gmtdefaults4.base .gmtdefaults4
3
4  OUT=psxy1.ps
5  PRO=-JX15/10
6  REG=-R0/10/1/8
7  ANN=-B1/0.5
8
9  psxy $REG $PRO $ANN <<END > $OUT
10 0 1
11 1 2
12 2 5
13 3 4
14 8 8
15 10 7
16 6 2
17 END
18
19 gv $OUT
20 ps2raster -A -Te $OUT
21 rm $OUT

```

- The `<<END` in line eight signifies 'reading standard input till `END` (in line 16) is reached'. Important:
 - The final string (here `END`) must always start in the first column.
 - After the final string must be line break (and no blank character or `<TAB>`).
 - The string `END` is arbitrary, every random string can be used.
- The command `ps2raster -A` is useful to remove the white frame around the postscript-plot. That might make sense when the plot is to be used in another document (e.g. LaTeX). The option `-Te` creates an `eps`-file, other options are e.g. `jpeg` or `png`. Further information can be found in the `ps2raster-manual` man `ps2raster`. Examine the difference with `gv`!

Figure 3.1 shows the plot from Script 3.1 and what can be achieved by adjusting some options and parameters (see Exercise 2).

Example 2

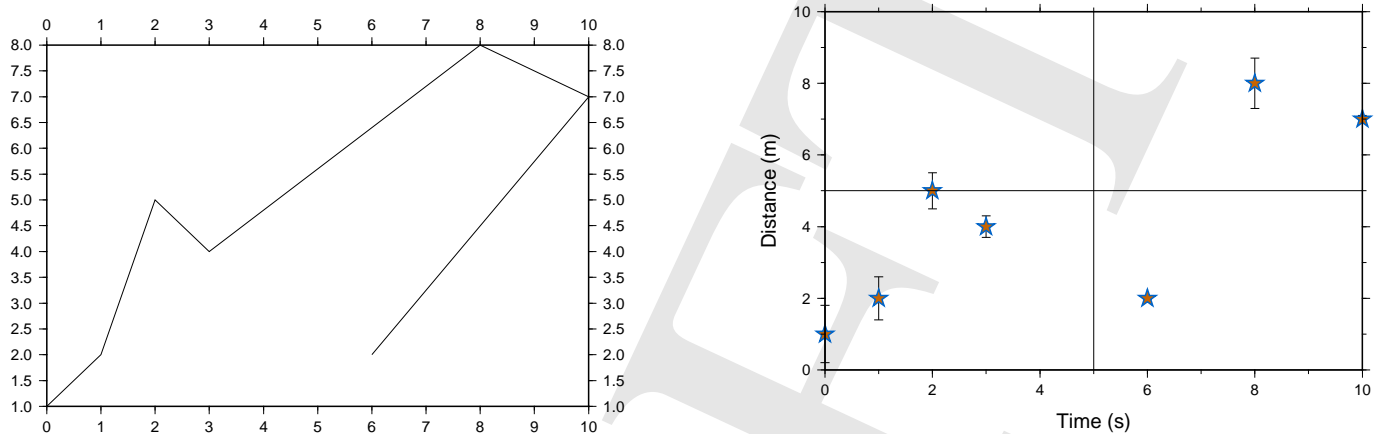


Figure 3.1: Plot from Script 3.1 before (left) and after (right) changing some options and parameters.

3.2 Exercise 2: `psxy`, axis labels, color

1. Copy the Script 3.1 with the command


```
cp ~/de.awi.GMTCourse/in_psxy.sh .
```

 and execute it.
2. The option `-J` defines the projection method, the `X` stands for a linear projection. The two following numbers set the size of the plot in cm.
 - (a) Modify the numbers slightly.
 - (b) Remove the second number and the `/`, what does it mean?
3. Try to find out what's the meaning of parameter `-R` and modify it.
4. The option `-B` is quite powerful and therefore quite complex. To check out the capabilities of this option try the following modifications:
 - (a) Slightly modify the two numbers.
 - (b) leave out the second number and the `/`.
 - (c) Try the following: `ANN=-B1/1SWne`. The letters represent the four cardinal directions.
 - i. Switch the capital and the small letters.
 - ii. Completely remove single letters.
 - (d) Frame-tics and gridlines can be created with `f` and `g`. Test `ANN=-B1f0.5g2/1f0.75g5SWne` and change the values of the parameters until the result is satisfying.
 - (e) Now add axis labels `ANN=-B1:x-axis:/1:y-axis:SWne`
 - (f) and a title `ANN=-B1:x-axis:/1:y-axis:..Example2:SWne`.
 - (g) All former labels did not include any blank spaces. But at least in the title a blank space in front of the 2 would make sense. To achieve that two points have to be taken care of:
 - i. The whole title must be set in (single or double) quotation marks. Otherwise the variable `ANN` is only assigned until the first blank space and the rest can not be interpreted by the shell.


```
ANN="-B1:x-axis:/1:y-axis:..Example 2:SWne"
```
 - ii. To make GMT read the whole option `-B` (not only till the first blank space following `Example`) the variable `$ANN` must be set in *double* quotation marks: `psxy $REG $PRO "$ANN"<<END>$OUT`. Single quotation marks would prevent the shell from interpreting `$`-sign as the *value of ANN* – the shell would just submit the `$ANN` to GMT.

A necessity for blank spaces exists in axis labelling when it comes to units, experiment with it!

5. The read data pairs are connected with a line as standard option, but `psxy` offers a great number of options to modify the representation of the data:
 - (a) Add the option `-L` to force the representation as a closed traverse.
 - (b) Replace the `-L` by `-Sc0.3` (`S` stands for *symbol*, `c` für *circle*).
 - (c) Add `-G150` (color of the points).
 - (d) Add `-W5` (lines around the points), try to find out what the number behind the `-W` does.
 - (e) Replace `-Sc0.3` by `-Sa0.5` (the `a` stands for *star*), what does the number mean? Modify it!
 - (f) Add a `-N`, watch the boundary points.
 - (g) Read the section of the manpage of `psxy` that deals with the symbols. Try other symbols. (Attention: some symbols need more than two data rows.)
 - (h) Add an error estimator (in y-direction).
 - (i) For the black and white representation GMT uses grey-scale values from 0 (black) till 255 (white). For the representation of colors the RGB color model is used. In this model each of the colors red, green and blue is assigned a number between 0 and 255.
 - i. Change the color of the symbols with the option `-G` e.g. to `-G200/100/0`.
 - ii. Change the color of the lines (option `-W`), e.g. to `-W5/0/100/200`.

With support from Figure A.15 on page 40 it should be easy to work with RGB color model, try other colors!

3.3 Reading input from a file

An important premise for reading data from a file is the existence of one. In Script 3.2 two files each with two rows are created with `gmtmath`. `gmtmath` is a calculator that works with *Reverse Polish Notation*. More information can be found in the manpage of `gmtmath` (`man gmtmath`). Here it is sufficient to understand that the x-axis ranges from 0 to 100 and that the square root of x is calculated at equally distributed sampling points with a step size of 10 and 1.

Script 3.2 `psxy`, `gmtmath`, input from a file

```

1 #!/bin/bash
2 cp gmtdefaults4.base .gmtdefaults4
3
4 OUT=psxy_datei.ps
5 IN1=psxy_datei_a.dat
6 IN2=psxy_datei_b.dat
7 PRO=-JX15/10
8 REG=-R0/100/0/10
9 ANN="-B10:x:/1:sqrt(x)::Beispiel_3:SWne"
10
11 gmtmath -T0/100/10 T SQRT = $IN1
12 gmtmath -T0/100/1 T SQRT = $IN2
13
14 psxy $REG $PRO "$ANN" $IN2 -W5/0/0/200t20_10:0 -K > $OUT
15 psxy $REG $PRO $IN1 -St0.3 -N -G200/0/0 -W5/0 -O >> $OUT
16
17 gv $OUT
18 rm $OUT $IN1 $IN2

```

As it is not possible to plot lines and points in *one* single `psxy` command, `psxy` is called twice with different options. Most options of the command `psxy` have already been discussed in Script 3.2 and Exercise 2, so only new options (and their parameters) are explained.

- The option `-W` plots a *solid* line as standard. To create a *dashed* line, the parameter `t` has to be added.

- The number of pixels to be drawn (here 20).
- The separator `_`
- The number of pixels *not* to be drawn (here 10).
- One :
- The number of pixels before the first pixel is drawn (here 0).
- The PostScript code created by the first `psxy` command in line 13 must not be *closed* (i.e. it must be written *no footer*). This can be prevented by the option `-K`.
- The PostScript code created by the second `psxy` command in line 14 must not
 - have a *header*. This is achieved by the option `-0`.
 - overwrite the output file, but *append* the code, therefore `>>` is used instead of `>`.
- The axes labels (`$ANN`) just need to be written once – so it is missing in line 14.

3.4 Exercise 3: `psxy`, `pstext`, special characters

1. Copy the Script 3.2 with the command


```
cp ~/de.awi.GMTCourse/in_psxy_file.sh .
```

 and execute it.
2. Permute the two variables `$IN1` and `$IN2` in the lines 14 and 15. Try to understand what changed (afterwards restore the original order.)
3. Remove the title.
4. Experiment with other settings for the parameter `t` in the option `-W`.
5. One of the most common mistakes working with GMT is the wrong usage of the options `-K` and `-0`, often in combination with mixing up `>` and `>>`. To be able to recognize the cause for the error containing (or missing `:-`) plot, the following errors are simulated:
 - (a) Remove the `-K` in line 13 (undo!).
 - (b) Remove `-0` in line 14 (undo!).
 - (c) Change `>>` to `>` in line 14 (undo!).
6. Add a text to the plot by working through the following points.
 - (a) The second `psxy` command must not write a footer.
 - (b) Add the following lines to line 15 in the script
 - i. `pstext $REG $PRO -0 <<END >> $OUT`
 - ii. `12 5 12 0 0 MC The root function`
 - iii. `END`
 You should understand the first and the third line (otherwise ask!). Bring to your mind what the additional information behind `pstext` means. (Why `-0`? Why no `-K`? Why `<<END>>`?) Execute the modified script, a text should appear in the plot.
 - (c) Open the manpage for `pstext` (`man pstext`) and try to find out what the seven parameters read via standard input mean.
 - (d) Change the position of the text so it starts at $(x,y)=(5,9)$, therefore *three* parameters must be changed.
 - (e) Try font No. 33 (but *never* hand in a plot with a calligraphic font otherwise loss of points is unpreventable ;-). All available fonts can be found in the in Section 1.2 mentioned *Technical Reference* Appendix G.
 - (f) Change the text color.
 - (g) Change the background color of the text.
 - (h) Try to add a second text line *without* using a new `pstext` command.
 - (i) The second line should contain the text „Kronecker symbol: δ_{ij} “ (refer to the *Technical Reference* Appendix F).

- (j) Experiment in a third line with the German umlauts. Use the following table (or look in the appendix of the GMT-cookbook):

Character	Character-encoding scheme		Character	Character-encoding scheme	
	ISOLatin1+	Standard +		Standard+	ISOLatin1+
Ä	\304	\276	ä	\344	\342
Ö	\326	\331	ö	\366	\363
Ü	\334	\335	ü	\374	\370
ß	\337	\373			

In the file `.gmtdefaults4` standard character-encoding scheme is defined. Since GMT4 β schemes can be used. With the command `gmtset CHAR_ENCODING = <Character-encoding-scheme>` the encoding-scheme can be changed. All octal codes can be found in the *Technical Reference* in Appendix F.

7. Compare Figure 3.2 with your own plot and try to find the cause for eventually existing differences.

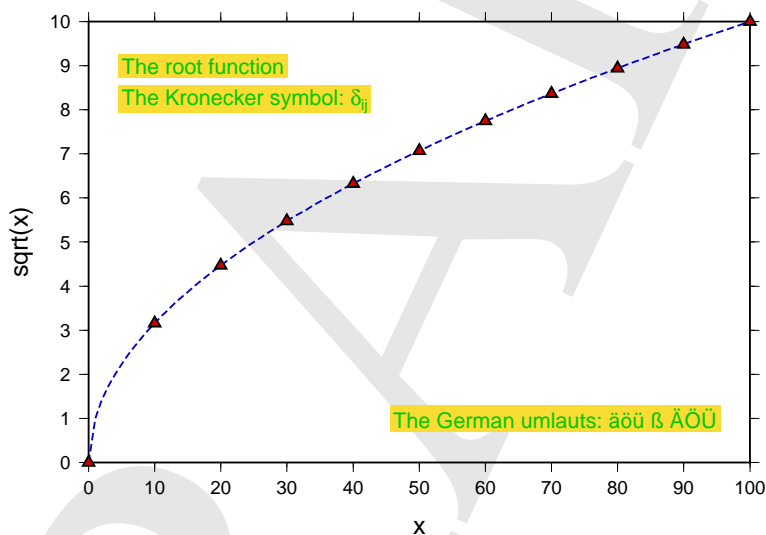


Figure 3.2: After completing Exercise 3 the figure for the modified Script 3.2 should look roughly like that.

4 Linux-tools & bash-programming

In most cases there is no file with just two columns which happen to be exactly the one you like to plot. So it is necessary to have a tool to choose the columns (and/or rows) to be plotted. GMT is *not* able to process anything else than the first columns. But that is not necessary anyway. It was a deliberate decision in the development of GMT not to program anything that other programs are already able to do. As two of the most important programs in this context `grep` and `awk` shall be introduced in this section. The most common usage of these programs (in connection with GMT) is demonstrated in Script 4.1.

- `grep` reads in a file (but is also able to read from standard input) and gives back only the rows matching a certain pattern. `grep` offers numerous options. In this case `-v` is used, which returns all lines *not* matching the pattern. A commentary line is conventionally marked by a `#`, the `grep` in line nine filters all the lines starting with that character and returns the other lines in standard output.
- The `|` is a *pipe*. That means that the output of the left command is used as input for the right command. In that case the result of the pipe is: `awk` receives the output of `grep` as input and `psxy` uses the output of `awk` as input. The line breaks after the pipes Script 4.1 are only reasoned by readability and are not obligatory.

Script 4.1 psxy, grep and awk

```

1  #!/bin/bash
2  cp gmtdefaults4.base .gmtdefaults4
3
4  OUT=grep_awk_1a.ps
5  IN=./data/grep_awk.dat
6  PROJ=-JX15/10
7  REG=-R0/635/0/0.7
8  ANN="-B100:Frequency_in_Hz:/0.1:Amplitude_in_@~m@~m:SWne"
9
10 grep -v '#' $IN |
11 awk '{ print $1,$4}' |
12 psxy $REG $PROJ "$ANN" -W5/200/0/0 > $OUT
13
14 gv $OUT
15 ps2raster -A -Te $OUT
16 rm $OUT

```

- `awk` is actually a powerful (script) programming language. In connection with GMT only few of its abilities are used. `awk` can read commands from a file, but in our case it is more effective to add the commands directly following the call of `awk`. Therefore, the commands must be enclosed in `'{...}'`. The most often used `awk` command is `print`. With option `$1` the first column is returned, with `$2` the second column and so on...

4.1 Exercise 4: psxy, grep, awk, logarithmic projection

1. Create the subdirectory `data` and copy the file containing the data:
`cp ~/de.awi.GMTCourse/data/grep_awk_1.dat ./data/.`
 Take a look at the data in the file (use `joe` or `emacs` or `less` or `cat` or ...).
2. Copy the Script 4.1 with the command
`cp ~/de.awi.GMTCourse/grep_awk_1a.sh .`
 and execute it.
3. Sometimes the linear projection is not the most suitable one. To use a logarithmic projection, the following changes have to be made:
 - (a) The projection has to be changed: `PROJ=-JX15/10l`. The `l` effects that the y-axis uses a logarithmic scale. Try to execute the script after this change and find out what causes the error!

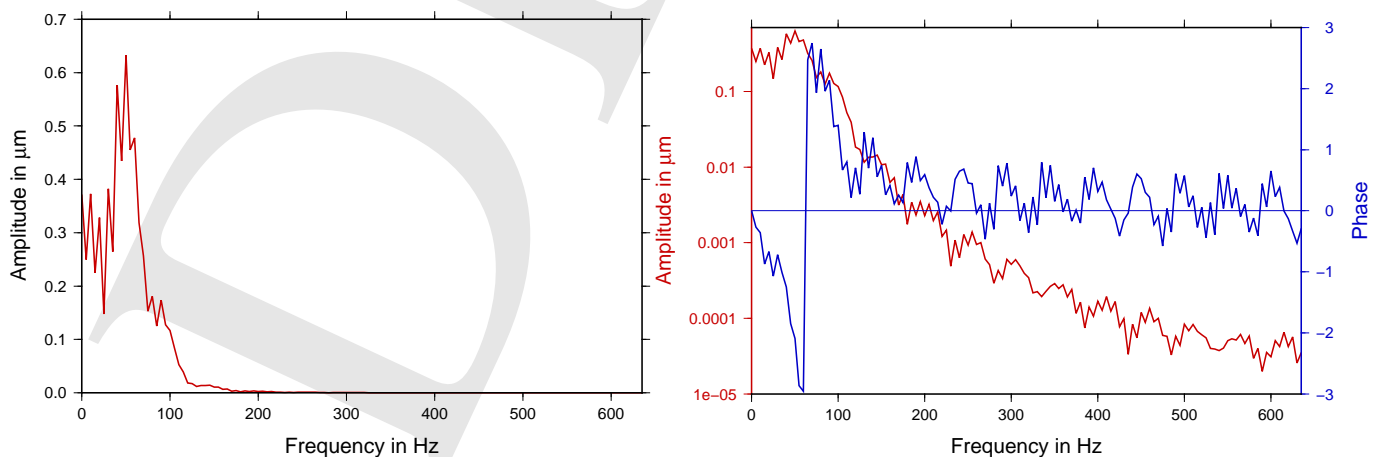


Figure 4.3: Example for Script 4.1 (left) and Exercise 4 (right).

- (b) Change the region option to `REG=-R0/635/1e-4/0.7`. Why is the amplitude of some of the frequencies missing? Change the region so that the amplitude can be plotted for all frequencies.
- (c) For caption of the axis in a logarithmic projection only the values 1, 2 or 3 should be used. (That means the value 0.1 of the option `-B` has to be replaced.) Test all three values for the y-Axis. For using „3“ you will have to change the height of the plot to be able read the caption. What is the effect of the three possibilities?
- (d) Try to use a logarithmic scale for the x-axis as well (undo!)
4. Sometimes it makes sense to write two graphs in one figure. That requires the following steps:
- (a) The first `psxy` command must not write a footer.
- (b) The phase is in the 5th column in the data file. Plot this column in another color. (You have to use `psxy` in combination with `grep` and `awk`.)
- (c) As it does not make sense to use a logarithmic scale for the phase, you have to define a new projection variable for the `psxy` command that does not contain 1.
- (d) You will see that you need another region for the y-axis. Add a second variable for the region. As it is in general, the region to be plotted is not known. Enter the following line in a shell (not the script):

```
grep -v '#' data/grep_awk_1.dat | awk '{print $1,$5}' | minmax
```

The GMT-command `minmax` returns the minimum and the maximum of all entered columns. It is possible to format this output in a way it can be used by `psxy` as region option:

```
grep -v '#' data/grep_awk_1.dat | awk '{print $1,$5}' | minmax -I1/1
```

This output is exactly what we want to write into the second region variable. In the shell script it can be used like that:

```
REG2='grep -v '#' data/grep_awk_1.dat | awk '{print$1,$5}' | minmax -I1/1'
```

Important are the two *reversed* single quotation marks at the beginning and the end of the command whose result is to be written into the variable. This quotation mark can be created (in `joe`) by pressing (maybe twice) the key combination `SHIFT + <key between ⌘ and Backspace>` on a German keyboard.
- (e) The caption of the axis has to be changed for the phase. Define a second variable that plots the caption of the axis on the right side and use this variable as option for the `psxy` command that plots the phase. Compare your plot with figure 4.3.
- (f) To plot the caption of the axis in the same color as the graph, one parameter in the file `.gmtdefaults4` has to be changed. This can be done from inside the script by using the command `gmtset BASEMAP _FRAME_RGB = 200/0/0`. `BASEMAP_FRAME_RGB` is the parameter that needs to be altered and `200/0/0` is the corresponding color. (A list of all parameters in `.gmtdefaults4` can be called by `man gmtdefaults`.) To create the three different colored captions of the axes in Figure 4.3 (right), three commands with each a certain option `-B` are necessary. The two y-axes can be created simultaneously with the corresponding plot, the caption of the x-axis either by a `psxy` command that receives no data via standard input or (simpler) with the command `psbasemap`.
5. `awk` can do much more: change the `awk`-command reading in the amplitude to `awk '{print $1,sqrt($2*$2+$3*$3)}'`. (Remember: The second column contains the real part, the third one the imaginary part). The same plot should appear. If the plot has changed, comment out the following pipe and the `psxy` command and take a look at the output of the two different `awk` commands via standard output. If you find a difference think about possible reasons. If you are not sure, ask, this is important! The solution to this problem (and many other problems as well): Every script using `awk` should contain the command `LANG=C` directly behind the first line.
6. Change the `awk` command reading the phase: Do not use the fifth column but calculate the phase from the real and imaginary part by using the function `atan2(y,x)`.
7. Write the `awk` command `if(i+%5==0)` in front of one of the print commands. What is changing (and why)?

4.2 Some hints for bash programming

As complex programs without any errors are hardly written on the first go, it is of advantage to test a code every time after implementing an extension. In many cases a syntax error is the cause for an abnormal termination. The following hints should make it possible to locate the error and correct it.

- Often it is useful to check if a variables value is what you expect it to be. This can be done with the command `echo $OUT`.
- The command `exit` terminates a `bash` script at any given point.
- Single lines can be commented out by using the `#`.

If you have knowledge in procedural programming¹, you will be familiar with the concepts introduced in this section and you will just have to get used to the syntax.

4.3 Functions and queries in the bash

A strength of programming languages is the possibility to perform similar tasks with slightly changed parameters with minimal effort. An example (though not the best) is the script of the last exercise, where the program `psxy` is called twice with preceding `grep` and `awk`. Even though it is not necessary in this case, that script can be used to show how such a structure is realized. Script 4.2 shows a `bash` script, that contains two functions. The essential parts of this script shall be explained in this section.

- Copy the Script 4.2 with the command `cp ~/de.awi.GMTCourse/in_bash.sh .` and execute it.
- Functions are initiated by the keyword `function`, followed by the name of the function and the (optional) parenthesis `()`. The body of the function is bordered by curly brackets `{...}`.
- The functions must be located ahead of of the main program. In Script 4.2 the main program only contains the definition of some variables, the call of the function `plot_ps`, the command to display the plot `gv` and the command `rm` to remove the created file.
- Variables declared within a function should be (if possible) `local` to prevent side effects in other parts of the program.
- One disadvantage of Script 4.2 is the necessity to use `psxy` to create an `empty` figure containing just a header.
- In line 14 a loop is started that ends in line 20. The loop is run twice, one time variable `i` is assigned the value `Amplitude`, the second time the value `Phase`.
- In every run of the loop the function `set_parameter` is called at the beginning, then the defined parameters are used to
 1. set the parameter `BASEMAP_FRAME_RGB` with the command `gmtset` to the value of `$COLOR`,
 2. create a temporary file, that contains just the two columns of the input data which shall be plotted,
 3. set the region option with the `minmax` command and
 4. create the plot by using `psxy`.

The only thing new (besides the loop itself) is the usage of the variables in combination with `awk`: Before the command (enclosed in `'{...}'`) the option `-v` is used to set the `awk` variable `c` to the value of the `bash` variable `COLUMN`. This is necessary because within the `awk` commands the (global) `bash` variables are not known.

- The function `set_parameter` writes the *one* parameter passed to it to the variable `WHAT`, this is just a matter of clarity.

¹A *procedural* programming language uses algorithms to formulate the necessary procedures. In general the following elements are used: variables, arithmetics, queries, loops and functions. Examples for this class of programming languages are amongst others Pascal, Basic, Fortran und C. Another approach is *modular* programming. Here, the focus is on subdividing and encapsulating the data in single modules. Examples are e.g. C++ and Java.

Script 4.2 psxy, minmax, bash programming, grep and awk

```

1  #!/bin/bash
2  LANG=C
3
4  cp gmtdefaults4.base .gmtdefaults4
5
6  OUT=bash1.ps
7  PRO=-JX15/10
8  IN=./data/grep_awk_1.dat
9  TEMP_DAT=tmp.dat
10
11 #####
12 function plot_ps()
13 {
14     psxy $PRO -R0/1/0/1 -K << END > $OUT
15 END
16     for i in Amplitude Phase ; do
17         set_parameter $i
18         gmtset BASEMAP_FRAME_RGB = $COLOR
19         grep -v '#' $IN | awk -v c=$COLUMN '{print $1,$c}' > $TEMP_DAT
20         local REG='minmax $MINMAX_INC $TEMP_DAT'
21         psxy $REG $PRO "$ANN" -W5/$COLOR -K -O $TEMP_DAT >> $OUT
22     done
23
24     local ANNAB=-B100:"Frequency in Hz":/Sn
25     gmtset BASEMAP_FRAME_RGB = 0/0/0
26     psbasemap $REG $PRO "$ANNAB" -O >> $OUT
27 }
28
29 #####
30 function set_parameter()
31 {
32     local WHAT=$1
33     if [ $WHAT == Amplitude ]; then
34         MINMAX_INC=-11/0.1
35         ANN="-B100:Frequency in Hz:W"
36         COLUMN=4
37         COLOR=200/0/0
38     elif [ $WHAT == Phase ]; then
39         MINMAX_INC=-11/1
40         ANN="-B/1g100:Phase:E"
41         COLUMN=5
42         COLOR=0/0/200
43     else
44         echo "error in 'select_for_gmt()'"
45         exit 1
46     fi
47 }
48
49 #####
50
51 plot_ps
52
53 gv $OUT
54 rm $OUT $TEMP_DAT

```

- By using an if-query the parameters are set. Important in this context is

- the semikolon ; before then has the same effect as a line break.
- The square brackets [...] must be surrounded by blank chars (or a line break).
- The == is a test-query, which is not to be mixed up with the assignment =.

4.4 Command line arguments for bash programs

Only if a program can be started with command line arguments it is really versatile. For example it might be helpful to have a program that writes a postscript header or footer depending on the command line argument. This program could replace the `psxy` command in line 12 of Script 4.2.

This program needs two values, the name of the postscript-file and the information if a header or a footer is to be written. Both could (theoretically) follow the call of the program (e.g. `write_head_foot.sh datei.ps F`) and then be read within the script with `$1` and `$2`. In this simple case that *might* even be sufficient. But already interchanging the arguments would result in a problem – and this method is totally inadequate if you need a script that reads an undertermined number of arguments (as every GMT command does).

Script 4.3 bash-program to write a header or a footer for GMT figures.

```

1  #!/ bin/ bash
2  #####
3  function usage()
4  {
5      echo -e "\n\nUsage: `basename $0` _has_to_be_called_with_\n\n\"
6              \"-O<PostScript-File> (output_file)\n\"
7              \"-w<[K|O]> (write_header(K)_or_footer(O))\n\"
8      exit 1
9  }
10 #####
11 function check_args()
12 {
13     OUT=NONE
14     HEADFOOT=NONE
15     FORCE=FALSE
16     while getopts fo:w: OPT ; do
17         case $OPT in
18             O) OUT=$OPTARG ;;
19             w) HEADFOOT=$OPTARG ;;
20             *) usage ;;
21         esac
22     done
23     if [ $OUT == NONE ]; then usage
24     elif [ $HEADFOOT != K -a $HEADFOOT != O ] ; then usage
25     fi
26 }
27 #####
28 function write_head_foot()
29 {
30     if [ $HEADFOOT == K ]; then
31         psxy -R0/1/0/1 -JX1 -$HEADFOOT /dev/null > $OUT
32     else
33         psxy -R0/1/0/1 -JX1 -$HEADFOOT /dev/null >> $OUT
34     fi
35 }
36 #####
37 check_args $*
38 write_head_foot

```

The `bash` offers an easy solution to this problem. The program in Script 4.3 is admittedly comparatively long, but in return it is robust and easily expandable (an advantage not to be underestimated). The new elements of this Script are explained in this section.

- Every program should contain a `usage()` function, that explains what it does and which options are offered.
 - The `-e` option of the `echo` is explained in the manpage (read it!).
 - Whereas `$1` and `$2` refer to the first and second argument passed to a `bash` program, `$0` contains the name of the program (including its path). The program `basename` removes all the path information from the file name. Take a look at the really short manpage!

- The `\` at the end of a line causes the program to ignore the line breaks, otherwise the `echo` command would have to be written in every line.
- As the function `usage()` is only called if an error occurs, it terminates the program with `exit`. The 1 behind `exit` is the *exit value* of the program. Conventionally a program returns the value 0 if it ran successfully and a number unequal zero if errors occurred. This exit value can be read by the parent program. What is the advantage of this concept? If you do not see it, ask!
- As first step the main program calls the function `check_args()` with the argument `$*`. That means all arguments are passed to the function. `check_args()` checks if the program has all the necessary arguments and if the parameters are valid. If anything is not correct, the function `usage()` is called and terminates the program.
 - At first the two variables `OUT` and `HEADFOOT` are initialized – otherwise there might occur problems with the `if` query. Try to understand what could cause these problems. If it is not obvious to you, run the program without it and/or ask.
 - The `while` loop executes the intern `bash` command `getopts` until all command line arguments are processed. The „-“ before the argument is removed and the rest is written to the variable `OPT`. The string `0:w:` indicates on the one hand that only the arguments `0` and `w` are valid, on the other hand (because of the „:“ that they need a parameter, which is saved in the variable `OPTARG`. (There are also arguments with no parameter, what is the difference?)
 - The `case`-query is an alternative (in this case a good one) to multiple `if` queries. Try to realize what is happening there and under which circumstances `usage()` is called.
 - At the end of the function it is checked if all conditions for a successful run are fulfilled. An output file must be defined (that is not called `NONE`) and the option `-w` must have been passed either the parameter `K` or the parameter `0`. The `-a` in line 23 represents the logical *and*.
- If no problems occur, the function `write_head_foot()` is called and writes the postscript header or footer. As `psxy` needs input data the (empty) file `/dev/null` is read in.

4.5 Exercise 5: Command line arguments for bash programs

1. Copy the Script 4.3 with the command


```
cp ~/de.awi.GMTCourse/tools/write_head_foot_start.sh ~/bin/.
```

 (Perhaps you will have to create the target directory with `mkdir` first.)
2. Make sure the directory `~/bin` exists and execute the script in an `xterm` (without options).
3. Execute the script twice, write a header in the first and a footer in the second run. Take a look at the result with `gv <file.ps>`.
4. Add an `optional` option to the program that decides whether an existing output file shall be overwritten when writing a header. The new option shall be addressed by `-f` (force).
 - (a) Supplement the function `usage()` first.
 - (b) Initialize the variable `FORCE` with the value `FALSE` in the function `check_args()`
 - (c) Modify the `while`-loop, so the call of the function with the option `-f`
 - i. does not cause a mistake,
 - ii. sets the value of the variable `FORCE` to `TRUE`.
 - (d) With the command `if [-e $OUT]` one can check whether a file exists. Add this test at an appropriate location and terminate the program with an explaining error message if the variable `FORCE` is not set accordingly.
5. Compare your program with the sample solution in Appendix C.1. This solution includes further expansions. Decide if you want to use this or your own solution for future work with GMT.
6. Change line 14 of the example Scripte 4.2 so it uses the the new program `write_head_foot.sh`.

4.6 Incidental remark

Now you know the basic GMT-commands (e.g. `psxy`, `pstext` and `minmax`), some useful Linux-tools (e.g. `grep` and `awk`) and the essential elements of `bash`-programming (e.g. loops, queries, functions, command line arguments, ...). With these tools it is possible to write complex programs for the visualisation of two-dimensional data with GMT.

The Additional Exercise 6 deepens the so far acquired knowledge of `bash`-programming with a challenging example. As the work through this exercise might take several hours and this workshop is focussed on the visualisation of data with GMT, you should only start this exercise if you have sufficient time or otherwise go forward to Section 5.

4.7 Additional exercise 6: bash-programming

This exercise is way more demanding than the ones you have done before. The `bash` script you have to write in order to create Figure 4.4 is about 100 lines long, so it may take a while until you get a satisfying result. But if you take your time to understand each step, you will be in the position to write `bash` scripts for complex tasks on your own. When done, you can compare your solution with the sample solution in the Appendix C.2. Do not get discouraged if an error message occurs instead of the expected result at the first go. In most cases a typing error, a missing blank char or a quotation mark too much or too less is the cause for the error. Do not be afraid to ask if you can not find an error or understand an idea.

1. Copy the file

```
cp ~/de.awi.GMTCourse/data/bash_task.dat ./data/.
```

Take a look at the file (it is best viewed with tabulator size 12, which can be set easily in `joe` by pressing `^T` and choosing the corresponding option with the cursor keys). This file contains the mean value, the quadratic mean value and the variance for each 20 (randomly chosen) points in time for different sample sizes. In this exercise a `bash` script is to be written that plots this data (compare Figure 4.4).

2. Open the (new) file `aufg5.sh` in an editor and take care that

- (a) the file is executed as a `bash` script,
- (b) the variable `LANG` is set to `C`,
- (c) the variable `IN` is defined and contains the path of the input file,
- (d) the variables `TDAT` and `TDAT2` define two temporary file names,
- (e) the variable `PRO=-JX15/6` is defined,
- (f) the variable `OUT` is defined and set to the name of the PostScript-file.

3. Save the script.

4. Make the script executable and test it (it should be executable without any error messages). While proceeding in this exercise execute the program after each change in the script. Syntax errors have to be corrected immediately (!), otherwise the program can not be developed in a rational manner.

5. As we want to have several plots in one PostScript-file, it is necessary to decide how header and footer of the PostScript-file shall be created. The easiest solution is to use the program `write_head_foot.sh` written in Exercise 5. Call it *twice*, once to write a header and once to write a footer. If an error should occur during the execution of the extern script (e.g. because the output-file already exists and no option `-f` was passed) the program should be terminated immediately. This can be achieved by testing the exit value of the extern script with the line

```
if [ $? -ne 0 ]; then exit 1; fi
```

(„\$?“ contains the exit value of the program called last and „-ne“ stands for *not equal*).

6. Take a look at the output of the script by adding the command `gv $OUT` and executing it.
7. Write a command that deletes the temporary files and the PostScript-file at the end of your `bash`-script.
8. Call the function `plot_all()` between the two calls of `write_head_foot.sh`.
9. Write the function `plot_all()`. It should contain

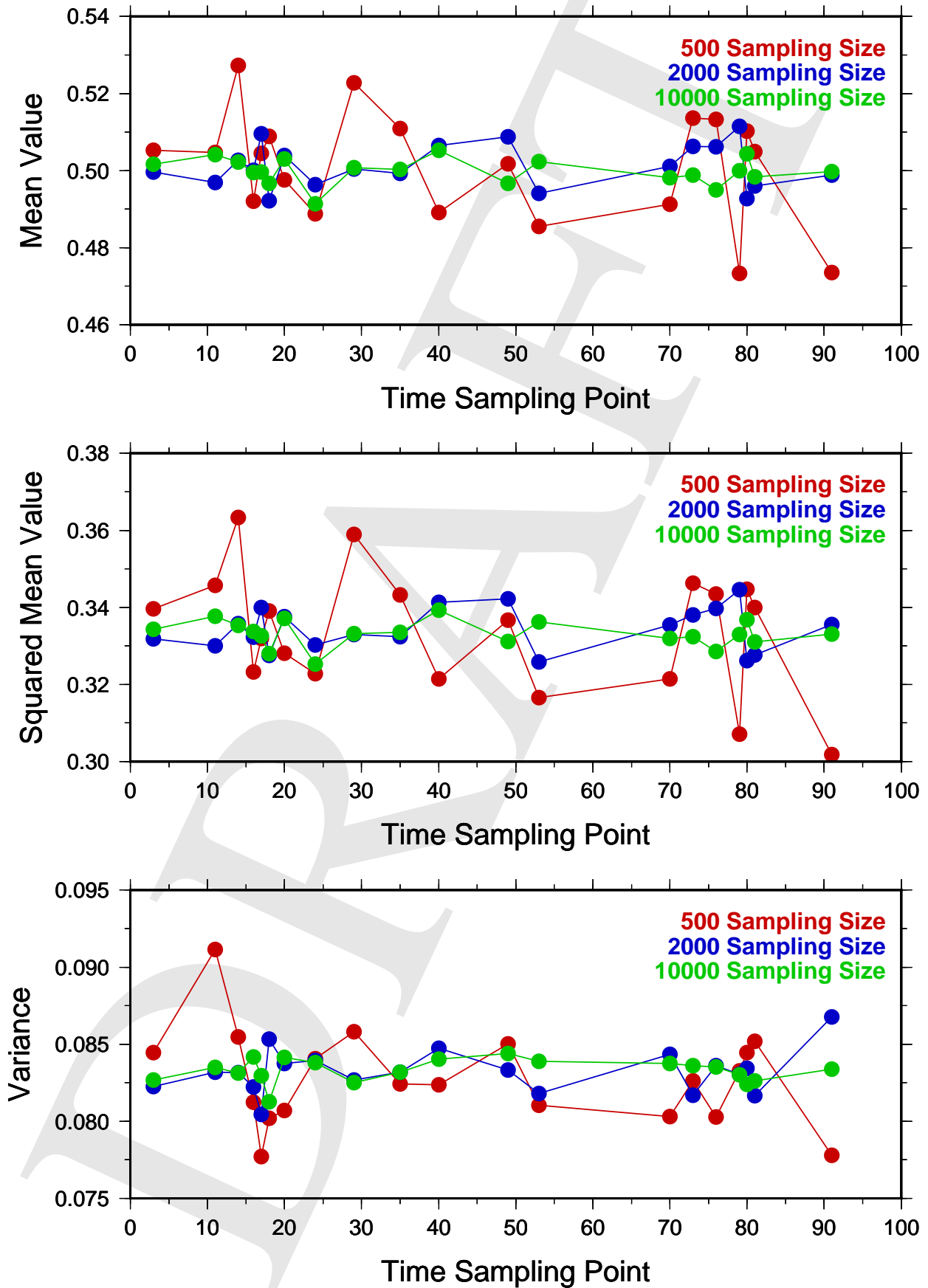


Figure 4.4: Plot for Exercise 6.

- (a) a loop over the three values M, QM and V.
 - (b) a call of the function `select_one()` with the current value of the loop variable.
10. Write the function `select_one()`, that accomplishes the following points
- (a) Save the passed argument in the variable `WHAT`.
 - (b) Depending on the value of the variable `WHAT` the variables
 - i. `COLUMN` (column of the input data),
 - ii. `ANNTXT` (label of the y-axis),
 - iii. `ANNINC` (increment of the y-Axis, use the value 0.2 to start with) and
 - iv. `OFFSET` (assign value 0 for M and 8.5 for QM and V. The meaning of this variable will become clear in the framework of this exercise.
 should be set in an if query. Remember to catch potential errors with an `else` query.
 - (c) Define the variable `ANN`:
 - i. The increment of the x-axis shall be 10.
 - ii. The label of the x-axis shall be `time sampling point`.
 - iii. For the y-axis the formerly defined values shall be used.
 - (d) Read the second, the one corresponding to the variable `COLUMN` and the third column from the input file and write the result in the temporary file `$TDAT`. Use the programs `grep` and `awk` (remember the `-v` option for the `awk` command!).
 - (e) Define the variable `REG` and use the program `minmax` with the option `-I1` to write the region option to it.
 - (f) Write a loop over the three values 500, 2000 and 10000 (the numbers correspond to the three sampling sizes). The loop body should contain the following commands:
 - i. An if query that sets for each of the three possible values of the loop variable
 - A. another color (e.g. `COLOR=200/0/0`)
 - B. the variable `OFFSET=0` for all but the first run of the loop.
 - ii. An `awk` command, that reads the lines from the file `$TDAT` for which the condition „third column of the line is equal to the string `"Size="<sample size>` “ is true and writes them into the file `$TDAT2`.
 - iii. The call of the function `plot_one()`.
11. Write the function `plot_one()`, that uses GMT-commands to plot the data in the file `$TDAT2`. The following commands should be used:
- (a) A `psxy` command, that plots the points in the corresponding color. Use the variables `ANN`, `REG`, `PRO` and `COLOR`. Attention: The variable `COLOR` contains just the RGB color code and (unlike the other variables) not a letter that defines the kind of option. Also, use the option `-Y$OFFSET`, `-K` and `-O`. Check if the script produces graphical output.
 - (b) You will notice that the scaling of the x-axis and y-axis is inadequate.
 - i. Modify the scaling of the axis by changing the parameter `-I` of the command `minmax` to `-I10/0.1`.
 - ii. It is better to choose the y-increment depending on the data. Define a new variable `YINC` at the right location and set it to an appropriate value.
 - iii. Create an axis caption with additional 'frametics' (compare Figure 4.4), by using `YINC` and `ANNINC` in the definition of `ANN`. (Attention: `a$YINCf$ANNINC` can not work because `f` is interpreted as part of the variable `$YINC`. To solve the problem use double quotation marks at the right location.)
 - (c) Now try to connect the points with a line by using a second `psxy` command. The result will not be sufficient.
 - i. Try to understand why; it might help to take a look at the file `$TDAT2`. That can be done e.g. by using the command `cat $TDAT2` at the appropriate location (here!) in the script (`man cat` explains the command). As it is enough to see the data once (from the first run of the loop), it makes sense to terminate the `bash`-program with `exit` afterwards.

- ii. You will realize the data is unsorted; the `psxy` command connects the data points in the given (and in this case pointless) order.
 - iii. To sort the data use the program `sort`. The output of `sort` can (and should) be directly passed to `psxy` via a pipe.
 - iv. Probably the plot is still not correct. To understand the reason take a look at the output of `sort`: The sorting is performed alphanumeric.
 - v. Open the manpage of `sort` and search for the option `-n`. Now you should be able to create the figure (compare Figure 4.4).
- (d) To change the order of the plots (the mean value first, the variance last) you have to
- i. change one loop
 - ii. modify the definition of the variable `OFFSET` at two locations.
- (e) Until now one can not see in the figure which color belongs to which sample size. Therefore the command `pstext` is used.
- i. Pass the sampling size to `plot_one()` and save it within the function to a new variable.
 - ii. The y-position of the text must also be passed to the function `plot_one()`, as it must not be constant (test it with a constant position first if you don't know why). Add the new variable `YTXTPOS` at the right location, pass it to `plot_one()` and change that function accordingly.
 - iii. Write a call of the `pstext` command, that reads in the text to be displayed via standard input (`<<END`).
12. After finishing this exercise your figure should look like Figure 4.4. Try to understand eventually existing differences.

5 Maps, cities, legends and more

With `pscoast` it is quite easy to generate maps in 25 different projections with rivers, political borders and so on. But in many cases it is necessary to add point data (e.g. positions of volcanoes, cities or deposits) with symbols and text as a second layer in such a map. How that works is explained in this chapter.

5.1 Exercise 7: Extern data, transparency, `pslegend`

In this exercise Figure 5.5 shall be generated. Though it is not really aesthetic due to too many colors and being overloaded, one gets to know some useful tools and options during its creation.

1. Write a `bash`-script that generates a map showing Germany and its political borders in Mercator projection. As region option use `-R4/18/45/56` (comp. Fig. 5.5).
2. GMT comes with a database of the political borders, but it can not assign closed traverses to countries. For this purpose, external data (e.g., from the internet) is needed. For the purpose of this workshop you'll find the necessary data in `textttde.awi.GMTCourse/data/germany2pts.txt`.
3. Take a look at the data with an editor. It consists of several segments (why?). GMT is also able to process `multi-segment` files. If you take a look at the manpage of `psxy` you will learn that therefore the option `-M` must be used and the segments are separated by a `>`.
4. Use this dataset to color Germany (compare Figure 5.5).
5. Plot the
 - (a) Water areas,
 - (b) rivers,
 - (c) state borders and
 - (d) a scale (try `gmtset LABEL_FONT_SIZE = xx` or `gmtset ANNOT_FONT_SIZE = xx` to set font size).

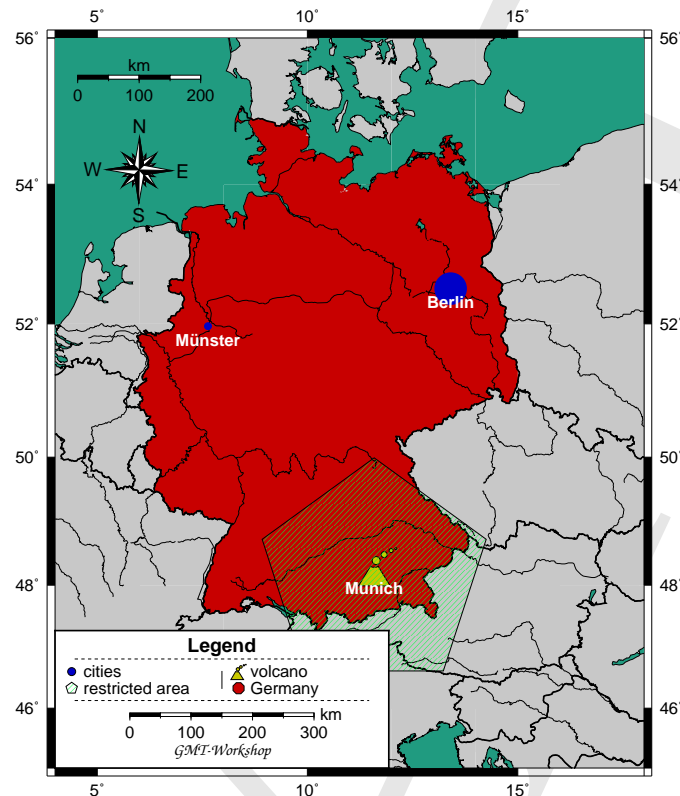


Figure 5.5: Examples for creative use of `pscoast`, `psxy` and `pslegend`.

(e) a directional rose

with *one* additional `pscoast` command.

6. To display cities in the map open a new file (e.g. `german_citys.txt`) and enter for the cities (e.g.) Münster, Munich and Berlin in the first column the longitude, in the second the latitude, in the third the name of the city and in the fourth the number of inhabitants. Instead of the German umlauts you have to use the corresponding octal codes (see p. 13).
7. Use `awk` and `psxy` to plot the cities as dots. Calculate the size of the dots automatically by a suitable scaling of the number of inhabitants.
8. Use `awk` and `pstext` to label the cities. To generate the necessary input for `pstext`, the command `awk '{print $1, $2-0.2,12,0,1,"MC", $3}' $INC` is quite handy (`$INC` is the variable where the input file `german_citys.txt` is saved). Try to understand (eventually consult `man pstext`) what's happening.
9. It is also possible to use your own (or predefined) symbols (defined by a polygon) by using the command `psxy -Sk`. A list with all predefined symbols can be found in `$GMTHOME/share/custom/`. For example you can create a volcano eruption in Munich by using the predefined symbol `volcano`. To read only the line with „Munich“ from the input file one can use the command `awk '{if($3 == "Munich") print $1,$2}' $INC` as filter.
10. Draw a „restricted area“ around the volcano eruption. Use the command `psxy` with the option `-Gp300/8:F0/255/64B-`. Read the manpage, to find out what `p300/8` means. Vary the values! Unfortunately the manpage is not complete. Behind the „:“ is declared which colors are used as foreground (**F**) and background (**B**). The `-` does not stand for a color, but a *transparency* (a markedly useful feature).
11. It was possible to generate legends with several `pstext` and `psxy` commandy. However, since GMT4 β the `pslegend` command makes life a lot easier. Take a look at the manpage (especially the given examples) and try to create

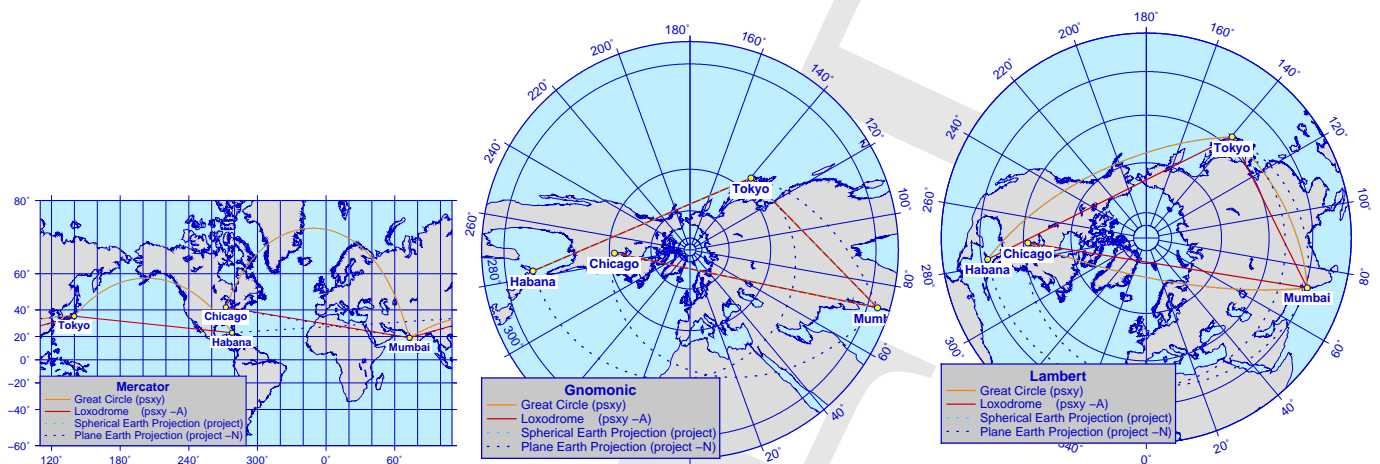


Figure 5.6: Examples for different projections and node connections on a sphere.

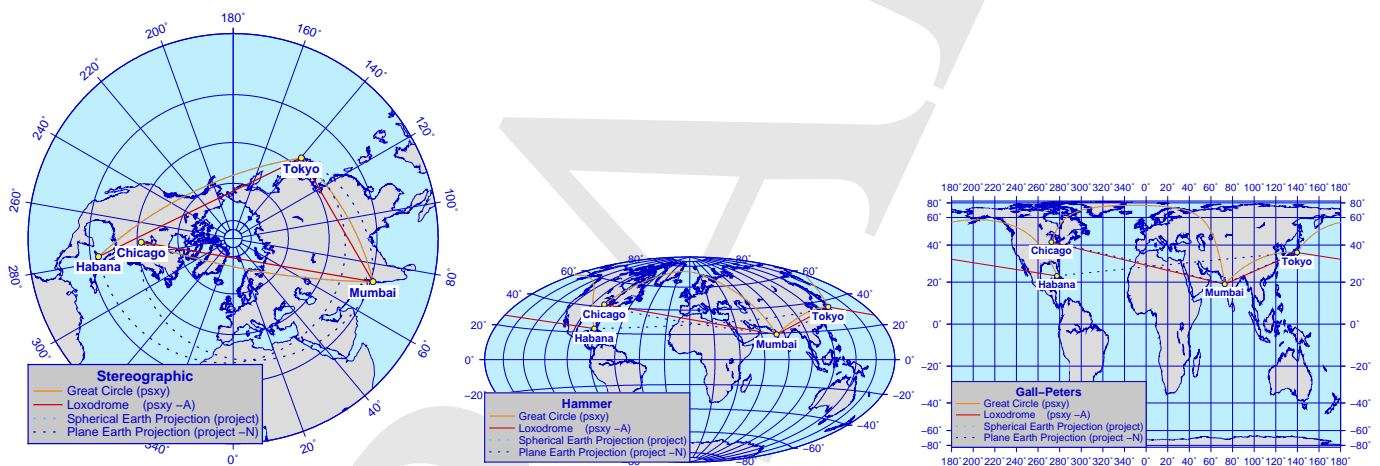


Figure 5.7: Additional projection examples.

a legend as seen in Fig. 5.5. To set the font size in the legend you can use `gmtset ANNOT_FONT_SIZE = xx`.

5.2 Exercise 8: Something about projections

GMT comes with many many projections. Not all projections are suitable for all tasks. Keep in mind, that the interpretation of geographical scientific data may depend on the projection!

1. Take a look at Figure 5.6 and try to understand the difference of the lines in connection with the applied projection.
2. Copy the following script to your working directory

```
cp ~/de.awi.GMTCourse/projections_start.sh .
```
3. Try to understand what the script does and what the different lines mean in each projection.
4. Adjust the script so that it shows the created map in the following additional projections: *Stereographic*, *Hammer*, and *Gall-Peters* (see Figure 5.7).
5. Note: The *one-world maps*, sold by many alternative stores, usually apply the Gall-Peters projection. Why?
6. Compare your solution with those in C.4.

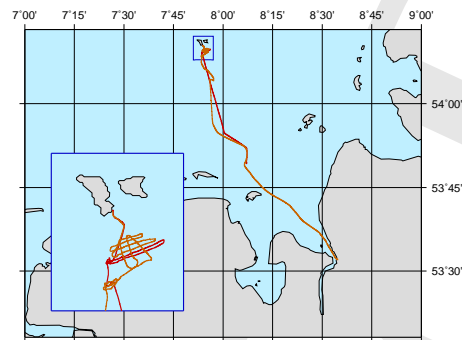


Figure 5.8: Plotting a ship track from GPS-data.

5.3 Exercise 9: Plotting ship track GPS-data on a map

We have introduced many tools to apply our knowledge to a simple application: Consider some files with given GMP-data. Your job is to visualize the track. In this example we have two GPS-files (one for each day) we will plot on a map of the German Bight.

1. Copy the following GMP-files to your working directory

```
cp ~/de.awi.GMTCourse/data/nmea.day[12] ..
```
2. Take a look at those files, this is the NMEA-output of a GPS. The longitude and latitude information we need has to be extracted from the lines starting with \$GPGGA. According to <http://www.gpsinformation.org/dale/nmea.htm>, the geographical information within the NMEA-record is coded as

```
4807.038,N Latitude 48 deg 07.038' N
01131.000,E Longitude 11 deg 31.000' E
```
3. Write a bash-script:
 - (a) Write a loop over both nmea.day files
 - (b) Extract the lines starting with \$GPGGA (hint: use `grep ^`)
 - (c) Use `awk` to extract the degrees and minutes (with `substr`) and calculate decimals from the minutes.
 - (d) Write the result into one or two files.
 - (e) Create a figure looking similar to Figure 5.8.
4. Many solutions are possible, compare yours with

```
~/de.awi.GMTCourse/nmea.sh.
```

5.4 psxy and the date format

A new feature since GMT4 are *time axes* in XY-plots. It is easy to use the time format in the ISO-8601 norm, but individual formats can be defined as well. Here one example for the definition of abscissa and ordinate:

```
-R/2001-01-01T00:00:00/2001-03-15T12:00:00/40/80 X-axis from 1.1.01 till 15.3.01, 12:00
-JX16.5T/3.0 Width of the time axis: 16.5 cm, height of the plot: 3 cm
```

The label of the axis can be set via `-B`. With `gmtset TIME_LANGUAGE <language>` it is possible to switch between different languages. This is illustrated in Script 5.1. `-B[p]` defines the *primary* label, `-Bs` the *secondary* label. In the upper part for example the first labelled interval is the year (`-1Y`), the second is the month, every third month is labelled in an abbreviated form (Jan, Feb etc.) (`30`), every month gets a frametic (`1o`) and every 12th (`12o`) is marked by a line. Please note that the y-axis caption for `-Bs` is not plotted (`/a0f0`).

An overview of the many date and time options for GMT4 can be found under http://gmt.soest.hawaii.edu/gmt4/doc/html/GMT_Docs/node22.html or directly by using `'man gmtdefaults'`.

Script 5.1 psxy with date format

```

1  #!/ bin/ bash
2  cp  gmtdefaults4.base  .gmtdefaults4
3
4
5  OUT=date_format.ps ; PRO=-JX16.5T/3
6
7
8  gmtset TIME_LANGUAGE us          # oben #####
9  gmtset PLOT_DATE_FORMAT o        # o: month only
10 gmtset TIME_FORMAT_PRIMARY Ao    # Ao: abbreviated month in capital letters
11
12 #X- Achse  Time
13 dat 11=2001-01-01T00:00:00
14 dat 12=2003-01-01T00:00:00
15 #Y- Achse
16 dat 21=40
17 dat 22=80.1
18
19 psbasemap -R$dat11/$dat12/$dat21/$dat22 $PRO -X3 -Y15 \
20 -B3Of1o:""/a20f10g20:"value":WSe -Bsa1Yg12o/a0f0 -K > $OUT
21
22 psxy -R $PRO -W7/50 -O -K <<EOF >> $OUT
23 2001-03-12 69
24 2001-07-02 66
25 EOF
26
27 gmtset TIME_LANGUAGE fr          # Middle #####
28 gmtset PLOT_DATE_FORMAT -"dd_o"  # replacing the leading zeros with '-'
29 gmtset PLOT_CLOCK_FORMAT hh      # hour only
30
31 dat 11=2001-01-01T00:00:00
32 dat 12=2001-01-05T00:00:00
33
34 psbasemap -R$dat11/$dat12/$dat21/$dat22 $PRO \
35 -B6hf1h/a20f10g20WSe -Bsa1Df1hg1d/a0f0 -Y-5 -K -O >> $OUT
36
37 psxy -R $PRO -W7/50 -K -O <<EOF >> $OUT
38 2001-01-03T06:41:00 53
39 2001-01-03T19:41:00 59
40 EOF
41
42 gmtset TIME_LANGUAGE no          # unten #####
43 gmtset PLOT_CLOCK_FORMAT hh:mm
44 gmtset TIME_FORMAT_SECONDARY fd  # fd: full name of day in small letters
45
46 dat 11=2001-02-02T00:00:00
47 dat 12=2001-02-04T11:00:00
48
49 psbasemap -R$dat11/$dat12/$dat21/$dat22 $PRO \
50 -B6Hf2h/a20f10g20WSe -Bsa1Kg1d/a0f0 -O -K -Y-5 >> $OUT
51
52 psxy -R $PRO -W7/50 -O <<EOF >> $OUT
53 2001-02-02T11:59:00 51
54 2001-02-02T14:59:00 71
55 EOF
56
57 gv $OUT &
58 ps2raster -A -Te $OUT
59 rm $OUT

```

The required settings are e.g. controlled by `gmtset TIME_LANGUAGE`, `PLOT_DATE_FORMAT`, `PLOT_CLOCK_FORMAT`, `TIME_FORMAT_PRIMARY`, or `TIME_FORMAT_SECONDARY`.

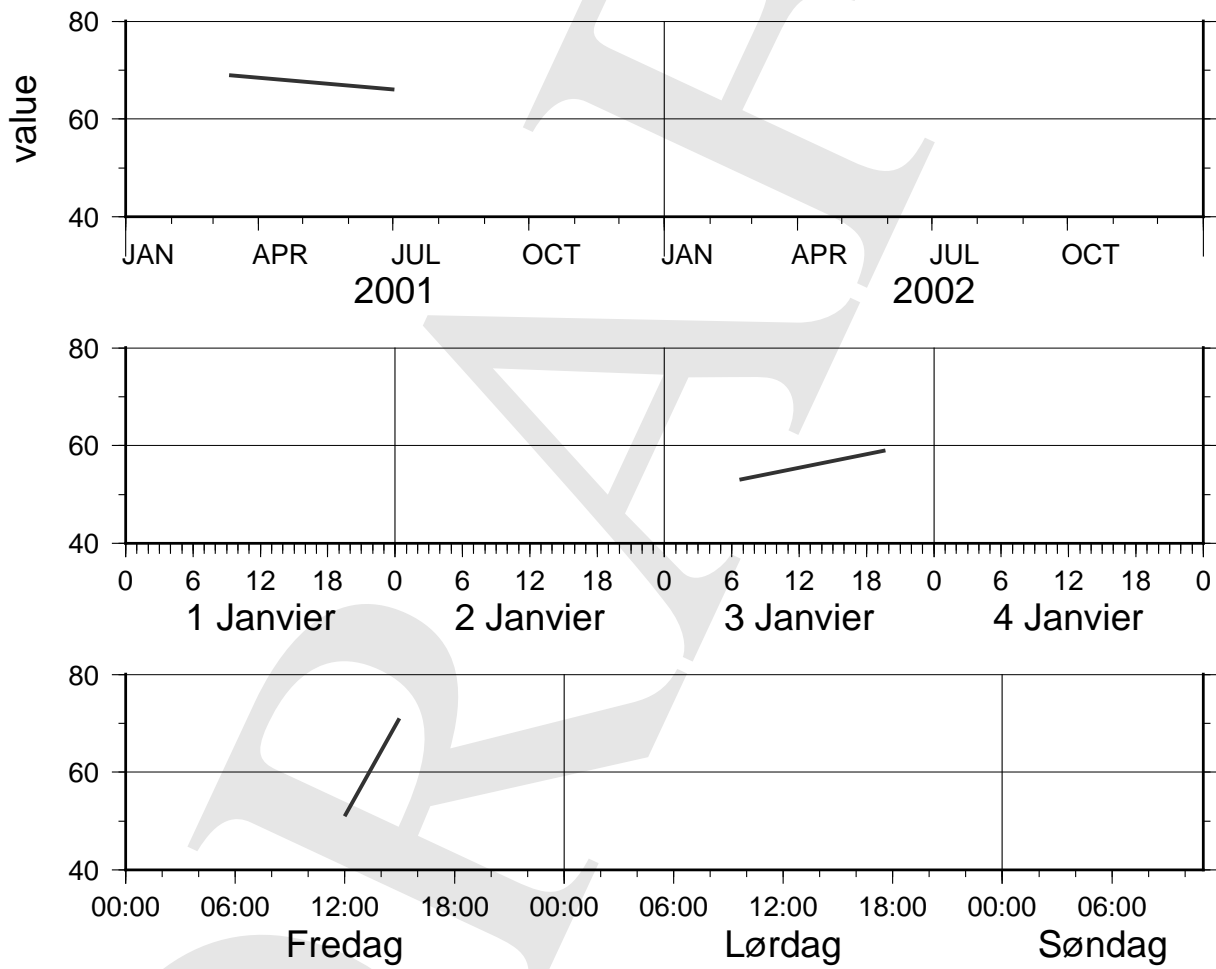


Figure 5.9: Three examples for date and time scales: Script 5.1.

6 Representation of data with two independant variables

For the representation of data that depends on two variable, GMT offers plenty of possibilities. In this chapter, the programs `psxyz` (Section 6.1), `grdcontour` (Section 6.3), `grdimage` (Section 6.5) and `grdview` (Exercise 13) are introduced. Finally the conversion of data to a format readable for GMT will be subject to Chapter 7.

6.1 Simple 3D graphs with psxyz

The most simple method to represent 3D data graphically is offered by the command `psxyz`. It is introduced in Script 6.1.

Script 6.1 psxyz

```

1  #!/bin/bash
2  cp gmtdefaults4.base .gmtdefaults4
3
4  OUT=psxyz_1a.ps
5  PRO="-JX15/10_-JZ5"
6  REG="-R0/10/0/8/0/10
7  ANN="-B1:x-axis:/1:y-axis:/2:z-axis:SWneZ
8  ANG="-E135/30
9
10 psxyz $REG $PRO $ANN $ANG -S0.4 -G0/0/200 <<END>$OUT
11 0 1 1
12 1 2 2
13 2 5 3
14 3 4 4
15 8 8 5
16 10 7 6
17 6 2 7
18 END
19
20 gv $OUT
21 ps2raster -A -Te $OUT
22 rm $OUT

```

The options of the commands `psxy` and `psxyz` do not differ substantially so in this section only new elements are discussed.

- The projection now occurs not only in two but in three dimensions. Therefore, additionally to `-JX`, `-JZ` is used.
- The region option `-R` is enhanced by `zmin` and `zmax`.
- The axis option `-B` now contains an additional third part for the z-axis.
- The biggest difference is the option `-E`, where the viewing angle is set by azimuth and elevation.
- The input data must (of course) consist of three columns (for some symbols even more).

6.2 Exercise 10: psxyz

1. Copy Script 6.1 with the command
`cp ~/de.awi.GMTCourse/psxyz_1a.sh .`
and execute it.
2. Vary the parameters of the projections `-JX` and `-JZ`.
3. Change the Z to z in the axis option `-B`, or completely delete this char. What happens?
4. Experiment with different viewing angels.
5. Test other symbols in different sizes.
6. Try to create the right graph in Figure 6.10. At first think about the differences between the left and the right graph. Use the manpage of `psxyz` and proceed step by step!

6.3 2D graphs with `grdcontour`

`psxyz` is unsuitable for the visualisation of 2D-fields, like elevation. Such data must be „gridded“ before it can be plotted. How to create grid files is subject to Chapter 7. This Section is about the visualisation of an existing grid file. The easiest possibility to plot 2D-fields is a contour plot. In Script 6.2 a simple example can be found.

Script 6.2 `grdcontour`

```

1 #/bin/bash
2 cp gmtdefaults4.base .gmtdefaults4
3
4 PRO=-JL10/43.5/35/50/15
5 REG=-R-10/30/35/59
6 ANN=-B10f5g5/5f5g2.5
7 OUT=grdcontour.ps
8 INGRD=./data/etopo5.grd
9
10
11 pscoast $REG $PRO $ANN -G200 -K > $OUT
12 grdcut $REG -Geurope.grd $INGRD
13 grdcontour $REG $PRO europe.grd -O -C500 -A1000 >> $OUT
14
15 gv $OUT
16 ps2raster -A -Te $OUT
17 rm $OUT europe.grd

```

- An Azimuthale-Lambert-Projection (-JL) is used. More information can be found in the manpage of `pscoast`, `psbasemap` or the official documentation of GMT (see Chapter 1.2).
- The variable `INGRD` contains the path of the file with the gridded data.
- The command `grdcontour` plots the contour lines.

6.4 Exercise 11: `grdcontour`

1. Get the topography from `ftp://ftp.awi.de/incoming/mthoma/etopo5.grd`
2. Copy the Script 6.2 with the command


```
cp ~/de.awi.GMTCourse/grdcontour.sh .
```

 and execute it.
3. Open `man grdcontour` and use it to solve the following tasks.
 - (a) Find out what is the matter with the two options `-A` and `-C` (with their current values) and vary their values.
 - (b) Add a yellow box as background for the caption.
 - (c) Add the option `-G` and try different parameters.
 - (d) Add a unit to the labels of the contour lines.
 - (e) Scale the data so the unit can be km.
4. Finally the plot should look like the central plot in Figure 6.11.
5. Now the script will be expanded, so that different regions can be plotted with the script.
 - (a) Create a loop over the values `Europa` and `Alpen`.
 - (b) Within the loop the two functions `select_region()` and `plot_ps()` shall be called.
 - i. Write the function `plot_ps()`. This function shall contain
 - A. the formerly used GMT commands,
 - B. `gv $OUT` and
 - C. `rm $OUT`.

(In this case `ps2raster` is not necessary.)

- ii. Write the function `select_region()`. In this function the variables `PRO`, `REG`, `ANN` and `OUT` shall be set in an if-query depending on the variable passed to the function. Use the following values for projection and region for the Alps: `PRO=-JC10/45.5/15` (Cylindric-Cassini-Projection) and `REG=-R5/15/43/48`.
- (c) The graph of the Alps should finally look like the right one in Figure. 6.11.

6.5 2D plots with `grdimage`

For the representation of contour lines, the abilities of `grdcontour` are obviously limited: There are areas with large and areas with small height gradients and therefore the density of contour lines varies strongly. More beautiful figures can be created by `grdimage`.

6.6 Exercise 12: `grdimage`

1. Enhance the script you wrote for Exercise 11 with the new function `check_args()`, where the arguments passed to the script are evaluated (compare Script 4.3 and Exercise 5).
 - (a) The Region shall be set with the Argument `-r` (`[E]urope`, `[A]lps`).
 - (b) The graphic representation shall be set with the argument `-r` (`grd[c]ontour`, `grd[i]mage`).
2. Initialize the variables in a way that a call of the program **without** arguments results in the figure known from Exercise 11.
3. Write an `usage()`-function that explains the supported arguments.
4. Test your program with different options.
5. Now add an if-query for the variable of the graphic representation to the function `plot_ps()`. Do not forget to catch potential occurring errors with `else`.
6. Add the following three lines to the part executed if `grdimage` is to be used
 - (a) `CPT=color.cpt`
 - (b) `makecpt -Ctopo -T-7000/3500/1000 > $CPT`
 - (c) `grdimage $INGRD $REG $PRO -C$CPT > $OUT`

and execute the script with the option to use `grdimage`; test both regions.

Probably you will be a bit disappointed by the result, but that can be changed. Anyway at first some basics shall be explained.

- (a) The program `grdimage` needs a colormap, that contains the information which values are assigned to which color. This colormap is read in by the option `-C`.
- (b) In this case the name of the colormap is saved in the variable `CPT`.
- (c) To create the colormap the command `makecpt` is used. The most important options for this program are:
 - i. `-T` sets the colors with the lowest and the highest value and defines the distance between the different colors
 - ii. `-C` is reference to a so-called `master-` colormap. The contained colors are scaled according to the option `-T`. GMT offers several pre-defined colormaps (Figure A.16), individual ones can be created if necessary.
7. Add a `pscoast` command that plots the coastline. As resolution choose *intermediate* and neglect structures smaller than 1000 km².
8. Add a legend for the used colormap with the command `psyscale`. Use the option `-D7.5/-1/15/0.5h -B1500:Topography:/:m:`
9. You will see that the plot is partially out of the visible domain. You can change that by adding a bigger offset in y-direction in the *first* GMT-command with the option `-Y5`.

10. Test the option `-I` and `-E` of the command `psscale`.
11. Test other values for the `-D` option of the command `psscale` (undo!).
12. Change the increment in the option `-T` of the `makecpt` command (e.g. 100 or 3000).
13. Test (with different increments) how a *continuous* color gradient affects the plot.
14. Test other colormaps, consult Figure A.16 for the selection.
15. Use the GMT-program `grdgradient` to increase the height of the structures in the plot:
 - (a) Define the variable `GRADGRD=gradient.grd`.
 - (b) The command


```
grdgradient $INGRD -G$GRADGRD -Ne0.6 -A0/270
```

 calculates the derivative in the directions defined by the option `-A` and normalized by the values set in the option `-N`. Further information can be acquired with `man grdgradient`.
 - (c) Add the created gradient file to the `grdimage` command with the option `-I`.
16. Execute the script. You will realize that it takes a long time. That is because the input file for the topometry/bathymetry contains the data for the whole earth. It would be absolutely sufficient to apply the command `grdgradient` (and all following) just to the domain that is going to be plotted. Therefore take the following steps:
 - (a) Define a new variable `INGRDCUT=in_tmp.grd`
 - (b) Open `man grdcut` and use it to find out how to cut out the target domain.
 - (c) Replace `$INGRD` by `$INGRDCUT` where necessary.
17. Test different parameters for the options `-N` and `-A` of the command `grdgradient`.
18. You will realize that the representation of Europe is quite good while the input data (with the 5 minutes resolution) is apparently not sufficient for the Alps. This problem can be solved with the command `grdsample`: Write an if-query that takes the following steps for the Alps:
 - (a) Defining the variable `INGRDCUT2=in_tmp_2.grd`,
 - (b) Calling the command `grdsample` and interpolating the topography in a resolution of one minute to the file `$INGRDCUT2` and
 - (c) Moving (not copying!) the file `$INGRDCUT2` to `$INGRDCUT`.

Attention: The command `grdsample` does not create any new data but interpolates between the existing data and creates new sample points. So the new result does not get any *better* but is just *more beautiful* (at least in most cases).
19. Use different colormaps depending on the region of interest.
20. By using the command `makecpt` one can abandon the option `-T` and the values in the master-cpt-file will be used as limits. Test that command with different colormaps!
21. Delete all temporary created files within the script.
22. Compare the plots your script generates with Figure 6.12.
23. Compare your solution with the sample solution in Appendix C.6.

6.7 Exercise 13: 3D graphs with `grdview`

A more plastic representation than with `grdimage` can be created with `grdview`. The options and parameters of both programs are pretty similar, that is why it only takes little effort to modify the solution of Exercise 12 (which can be found in Appendix C.6) so it uses `grdview`.

1. Enhance the description of the parameter `-g` in the `usage()` function, so it covers the use of `grdview` (parameter `v`).

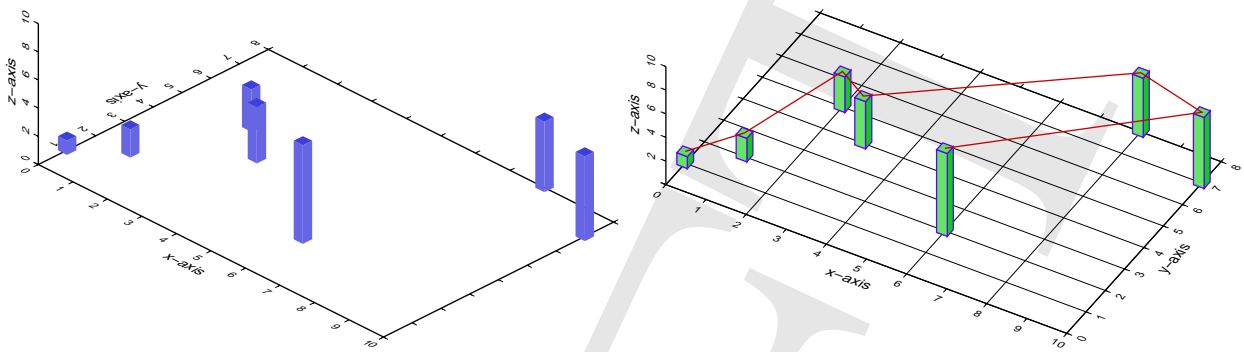


Figure 6.10: Example for psxyz and Exercise 10.

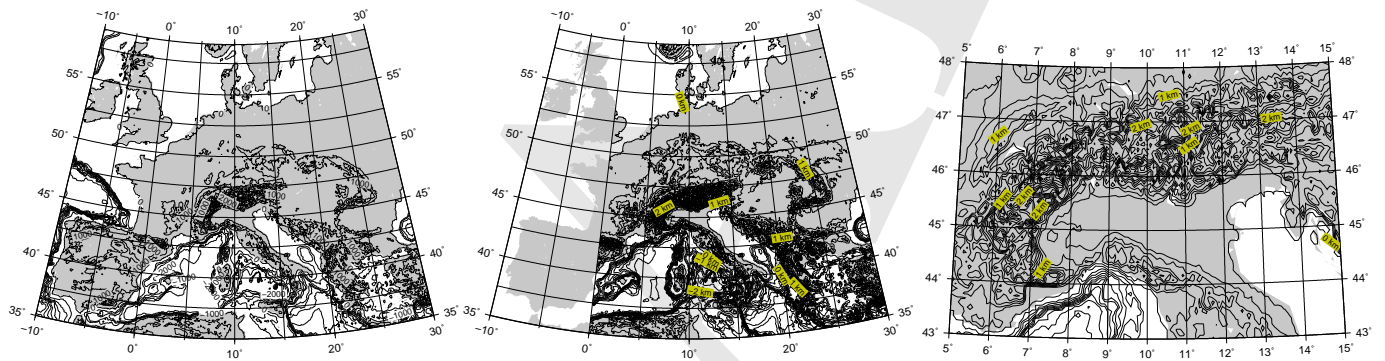


Figure 6.11: Examples for grdcontour and Exercise 11.

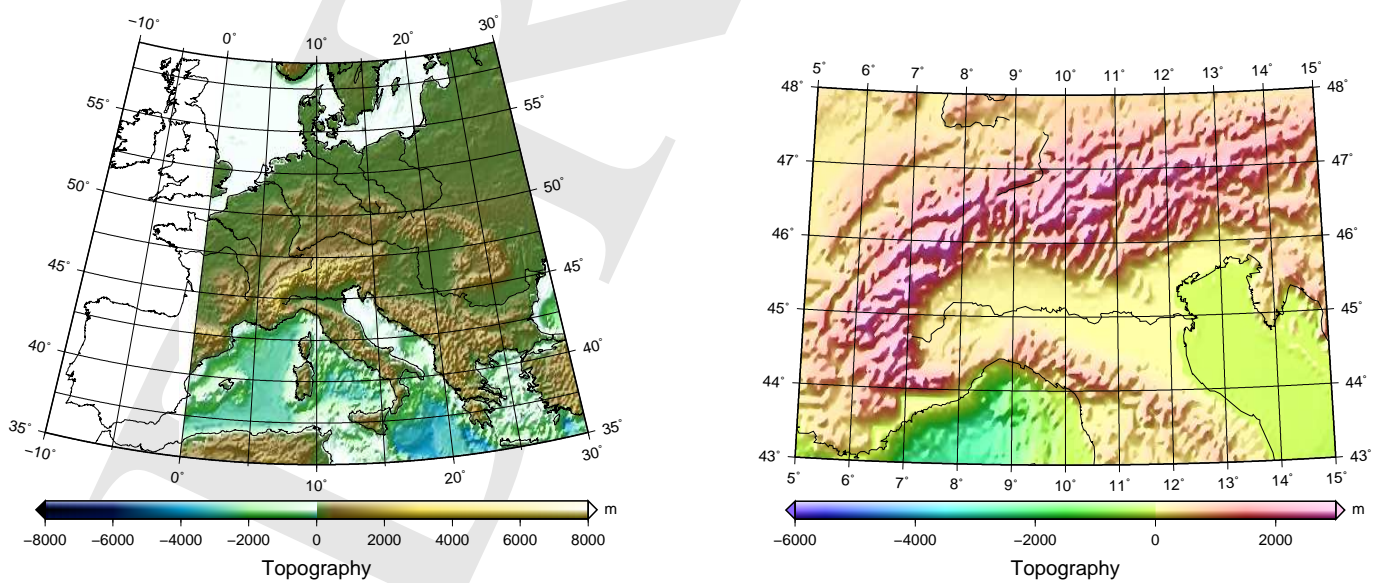


Figure 6.12: Topography of Europe and the Alps with grdimage.

2. Many commands in the the function `plot_ps` can be used for both, `grdimage` and `grdview`, therefore the implementation `elif [$SHOW == i -o $SHOW == v]; then` is reasonable.
3. `psscale` shall be called at last and write the footer.
4. Modify the program so the command line option `-gv` suppresses the execution of `grdimage` and `pscoast` but `grdview` is called. Additional to the options used by `grdimage`, `grdview` needs the following parameter: `-JZ3 -E200/40 -Qi`
 - (a) The options `-JZ3 -E200/40` are already known from Section 6.1 (`psxyz`).
 - (b) The option `-Qi` controls the style of the representation. More information can be found in the manpage (`man grdview`).
5. Now you should be able to generate the two plots in Figure 6.13 with your script.

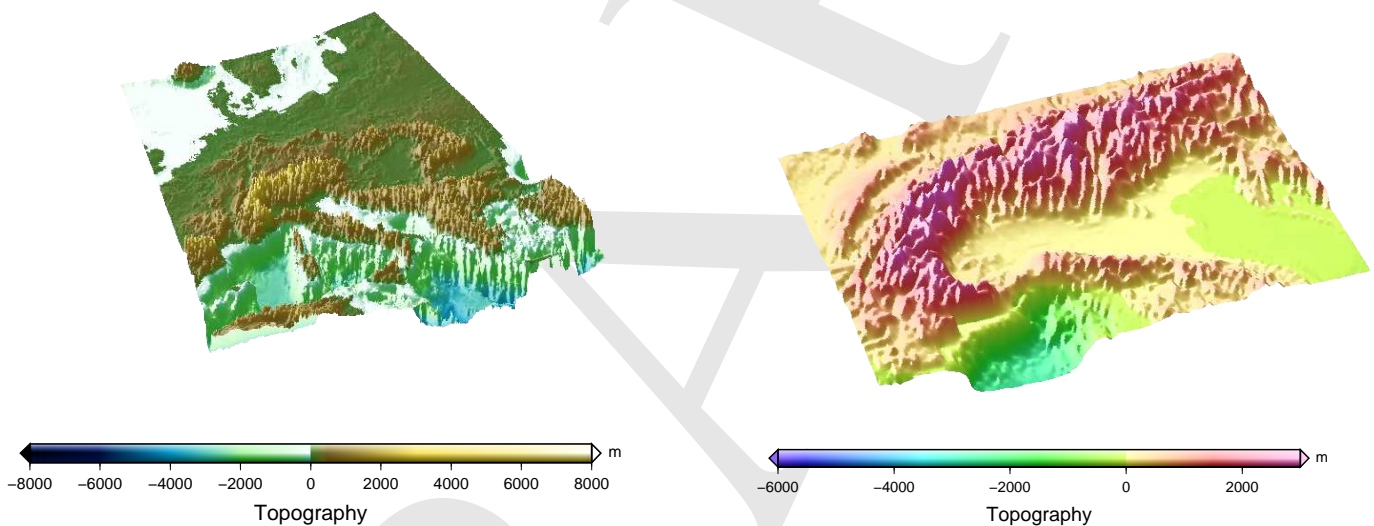


Figure 6.13: Topography of Europe and the Alps with `grdview`.

7 Creation of grid files in the netCDF-format

The programs `grdcontour`, `grdimage` and `grdview` are all reading in *grid files* as input data. A grid file contains all necessary information in *binary* form so the data can be saved compactly and processed quickly. To save these informations GMT uses the *netCDF* (Unidata Network Common Data Form) format². Not all sources (internet, research institutes, own measurements, ...) provide data in the netCDF-format, therefore in most cases a data conversion will be necessary. In the simplest case the data is available in a three-coloumned-table (as ASCII-file) with an *equidistant* (x, y, z) triple. If the domain is known (it can be get with `minmax`) and the constant distance of the data points as well, this data can be converted to the netCDF format with the command `xyz2grd -R<Region> -I<dx[/dy]>`. Binary data can also be converted to the netCFD format with `xyz2grd`, but in this case further information (in form of options) is required. These will be topic to Section 7.1.

If the data is *not* available equidistant, it has to be *gridded*. That means, the existing data is interpolated or extrapolated on an equidistant grid. This procedure is error prone as – depending on the used algorithm – the original information can be distorted (See Section 7.3).

²<http://www.unidata.ucar.edu/packages/netcdf/>.

7.1 Digital height models

A good topography data is available from the SRTM (Shuttle Radar Topography Mission, http://en.wikipedia.org/wiki/Shuttle_Radar_Topography_Mission). The data can be downloaded in different resolutions:

SRTM30: The SRTM30 dataset can be seen as the legitimate successor of GTOPO30. Both datasets have a resolution of 30" or round about 1 km. But the SRTM30 data is way more precise. That is mainly because the SRTM30 dataset contains only data from one source (radar antenna of the space shuttle), not to mention the better technology. All data south of 60°S and north of 60°N is more or less identical with GTOPO30, because there were no new measurements in this area. As the the dataset is quite large, it is divided in 28 parts.

SRTM3: The SRTM3 Data results from the same raw data as SRTMP30, but it has a resolution of 3" or round about 90 m.

The conversion of data to the netCDF-format is sometimes quite troublesome. Basically every data source requires different options and parameters. In the best case the dataset consists of a long list of height values, better binary than ASCII (much faster). For the conversion GMT provides the tool `xyz2grd`³. The documentation of the SRTM-data is supplied with the data. If you e.g. want to represent the Alps, the data of interest can be found in the file `e020n90`. Get the data (including the meta-information) from `ftp://ftp.awi.de/incoming/mthoma/e020n90.tar.gz` extract the file and take a closer look at the header:

```

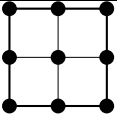
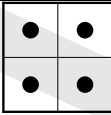
BYTEORDER      M
LAYOUT         BIL
NROWS          6000
NCOLS          4800
NBANDS         1
NBITS          16
BANDROWBYTES   9600
TOTALROWBYTES  9600
BANDGAPBYTES   0
NODATA         -9999
ULXMAP         20.004166666666667
ULYMAP         89.995833333333334
XDIM           0.0083333333333333
YDIM           0.0083333333333333

```

From these information the parameters for `xyz2grd` must be extracted. Open the manpage of `xyz2grd` for better understanding of the following information.

- `-R20/60/40/90`
results from content of the documentation.
- `-Ge020n90.grd`
Choose a name.
- `-I0.5m`
XDIM and YDIM set the resolution in x- and y-dicection. `0.0083333333333333` corresponds to 0.5 minutes.
- `-N-9999`
If there are regions without values in the dataset (e.g. sea) this value is assigned. In case of SRTM30 it is `-9999`.
- `-F`
Force Pixel Registration (Grid Registration is standard in GMT and does not need an option). Basically there are two possibilities to save extensive data: The data points can refer to the intersection of grid lines or just to the space in between them (see table). This is an important information for GMT as the number of sample points in rows and columns and therefore the overall number of data points is changed.

³If you have a *not documented* binary format that you want to convert - do not even think about it! (Except you have really much time and consider yourself being quite frustration tolerant :-)

	Grid Registration	Pixel Registration
graphic		
columns	$\frac{x_{\max}-x_{\min}}{I} + 1 = \frac{60-20}{0.008333333333} + 1 = 4801$	$\frac{x_{\max}-x_{\min}}{I} = \frac{60-20}{0.008333333333} = 4800$
rows	$\frac{y_{\max}-y_{\min}}{I} + 1 = \frac{90-40}{0.008333333333} + 1 = 6001$	$\frac{y_{\max}-y_{\min}}{I} = \frac{90-40}{0.008333333333} = 6000$

This is how one can see that the SRTM30 Data uses the Pixel Registration format and therefore the parameter -F is necessary. (The SRTM3 data uses the Grid Registration format!!!).

• -ZTLhw

If you try to take a look at the file `e020n90.dem` with an editor you will see pretty fast that it is a binary file. Basically the binary format is the best way to store large amounts of data as it can be saved with minimum space requirements and processed quite fast. Unfortunately, it is also a bit harder to convert. Generally, it is common that the data of a domain is written line by line from the top left to the bottom right corner. This is what the parameters **TL** stand for (**top-left**, that is default in GMT and can be dropped in this case).

The last two parameters are way more delicate. As you might know from programming C, Fortran or Pascal etc., there are a couple of different numerical types of variables which differ in the covered number range and precision. But they also differ in the required memory and here the trouble starts. `xyz2grd` needs to know what kind of number is used in the source file. In the manpage of `xyz2grd` is a list of all supported numerical formats. If working with DEMs you can always start with trying the parameter `h`. In programming C and C++ this is type `short`. It is meant to save integers with a relatively small range of numbers. Normally 2 byte are provided. Thus, for the coding of a short-value are 16 bit available ($2^{16} = 65536$ possibilities). One `unsigned short` is therefore able to code a number range from 0 – 65535. A `signed short` divides the number range equally in positive and negative values and lies consequently between -32.768 and 32.767 - virtually perfect for a digital height model.

If parameter `h` happens to be not working and `xyz2grd` results in an error message, the correct data type has to be calculated. Number of lines × Number of columns = Number of data points or $6000 \times 4800 = 28800000$. Now we take a look at the size of our DEMs (1s -1 `e020n90.dem`). The filesize is 57600000 byte. filesize / number of data points = memory per data point or $57600000/28800000 = 2$.

As you can see every data point in the SRTM30 dataset has 2 byte available. As it has to be possible to save negative height values one can logically conclude that the file format *signed short 2-byte integer* (or parameter `h`) must be used.

The parameter `w` (byte swapping) is necessary because most height models are created on Unix-workstations. The CPUs of workstations (big endian) save numbers consisting of several bytes in a different order than x86 CPUs of desktop computers (little endian).

7.2 Exercise 14: DEMs and xyz2grd

1. With the information of the last section, you should now be able to
 - (a) download the DEMs `e020n90.tar.gz` and `w020n90.tar.gz`
 - (b) convert them to the netCDF format and
 - (c) check the created `.grd` file with `grdinfo`.
2. Use the information contained in `grdcut` and `grdpaste` to generate a grid file for the region -R5/30/43/52.
3. Check the new grid file with `grdinfo`.
4. Another test of the grid file can be performed with `grd2xyz`. As the test of this high resolution dataset would take too long and this tutorial is mainly about the functionality of the programs, *resample* the dataset with `grdsample` to an interval of $1^\circ \times 1^\circ$ degree and use `grd2xyz` on the result. You should be able to interpret the output produced by the shell.

5. Plot the SRTM30-data with `grdimage`. Use the Lambert-projection `-JA18.5/47/15` and the region `-R10/43/28/51r`. Find out what the `r` stands for. Use one of the colormaps `GMT_topo.cpt`, `GMT_relief.cpt` or `GMT_globe.cpt`. These can be found in the directory `$GMTHOME/share/cpt/`. Use `grdgradient -A0/270 -Ne0.6` to create a gradient file and plot a legend with `psscale`.
Hint: As long as you experiment with the different parameters you should use the scaled-down grid file to decrease computing time. Figure 7.14 contains several examples which can be used as orientation. Note that the areas with no valid data (NAN, not a number) are masked in different colors. If a `*.cpt`-file does not contain any information for the color of NAN (a line starting with N), the value set in `.gmtdefaults` will be used. This value can be redefined with the command `gmtset COLOR_NAN = 0/50/150`.
6. GMT is also suitable to mask certain area of data.
 - (a) Use the dataset of the Austrian boarder `de.awi.GMTCourse/data/austria2pts.txt` for the next task.
 - (b) Use the command `grdmask` to create a new grid dataset. Assign 1 to all points inside (and on) and 0 to all points outside the Austrian boarders. The domain and the grid interval must be identical to the dataset which shall be masked. Try to understand the command
`grdmask austria2pts.txt -I0.5m -R5/30/43/52 -F -N0/1/1 -Gmask.grd`
 by reading the manpage.
 - (c) Use the command `grdmath` to multiply the two grid datasets:
`grdmath mask.grd <grid file>.grd MUL = austria.grd`
 Try to find out what this operation does. The command `grdmath` is quite powerful and enables numerous calculations, therefore a closer look at the manpage is recommended.
 - (d) Plot the new grid file `austria.grd` with `grdimage`.
7. Plot the SRTM30-data with `grdview`. Check out different `-Q` paramters. (Attention: `-Qc` creates a file three times as big as `-Qi` and the parameter `-Qs` makes this ratio even bigger then 75.)

A sample solution can be found in Appendix C.7.

7.3 Gridding of data

GMT offers three programs to grid data. In this context „gridding“ refers to converting unevenly distributed ASCII-data to the netCDF-format. The program `xyz2grd` does not belong to this category as it does *not* grid the data but just converts it from ASCII (respectively binary) to netCDF. As it would go far beyond the scope of this workshop to discuss the different gridding algorithms, this section shall just introduce the three programs provided by GMT. More detailed information and examples can be found in the Cookbook in the sections 7.12 and 7.14–7.16. Generally it has to be stated that there is *no* optimal gridding algorithm as the best choice of program strongly depends on the application and the existing data.

- **triangulate**

Gridding by Delauney-Triangulation: The existing (unevenly distributed) data points are connected with the aim of creating preferably equilateral triangles. This is achieved by maximizing the minimum angle in all triangles.

- A value is calculated for all points located inside a triangle.
- By using the distance to each corner of the triangle as a weighting factor these values can easily be computed.
- Points outside of the triangulated domain are not assigned any value (respectively NAN, in other words: There is no extrapolation).
- No point has a value smaller or bigger than the three local triangle points.
- The computed grid is *not* differentiable as points of discontinuity might arise where two neighbouring grid points are located in different triangles. That is physically not correct!
- By using the command `grdfilter` the field can be made differentiable via smoothening it.

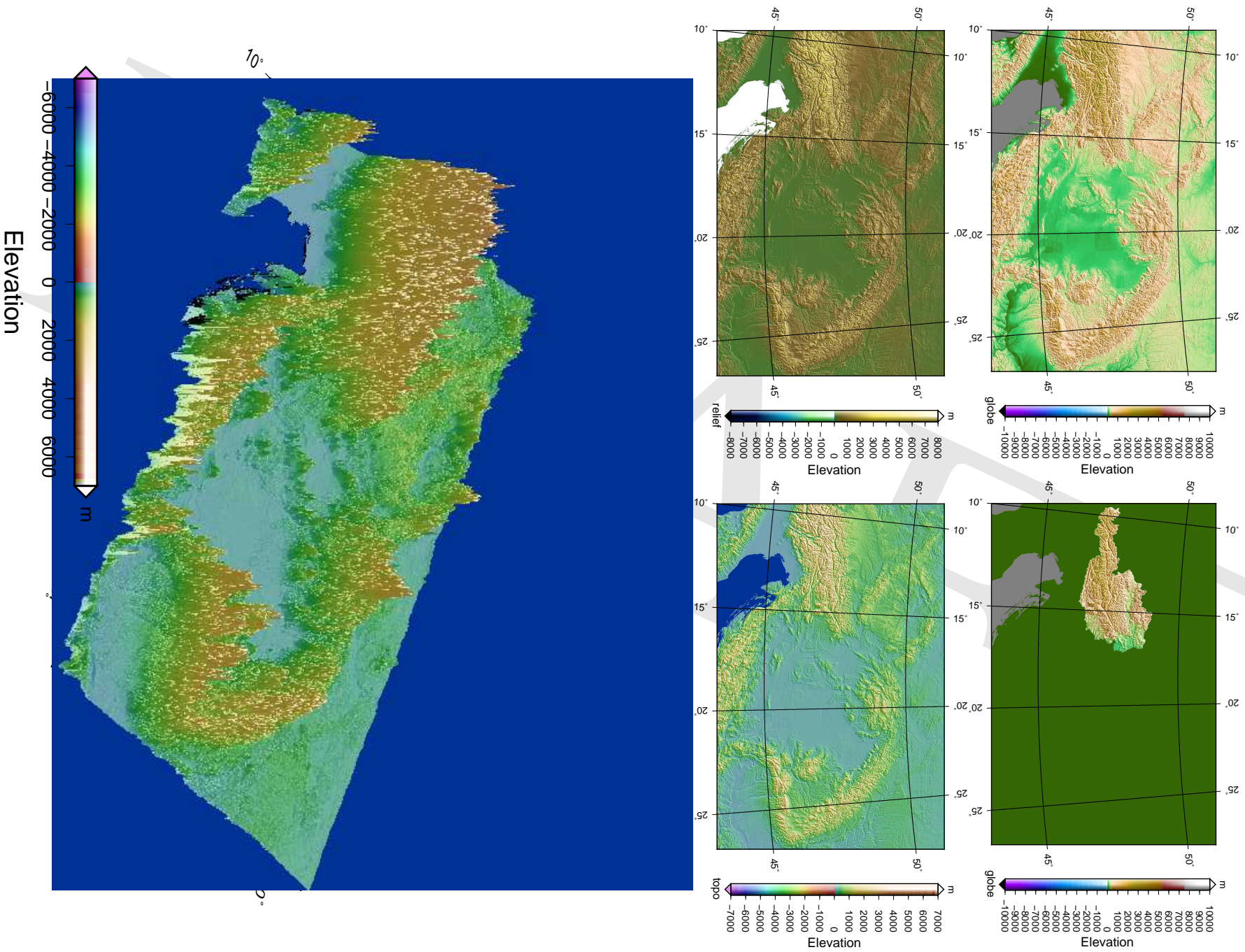


Figure 7.14: Graphic representation of the SRTM30-data (several examples).

- **nearneighbor**

For every point it is searched for existing data points in a given search radius `-S` that is divided in a certain number of directional sectors `-N`. From every sector the closest point is chosen and the value is calculated by a weighted mean value from all sectors.

- There is no extrapolation. Points with no valid data points in the search radius are assigned the value `NAN`.
- **nearneighbor** works best if the existing data points have about the same direction in x- and y-direction as otherwise the search radius would have to be a function of the search direction.

- **surface**

Is able to calculate good looking and smooth grids.

- For *every* point of the grid a value is calculated (extrapolation!).
- At the boundary points extremes might appear, especially if the distance to the closest existing data point is large.
- As an iterative method is used, the calculation of a grid with **surface** takes significantly longer than it would take with **triangulate** or **nearneighbor**.
- Before using **surface** one should use **blockmean**, **blockmedian** or **blockmode** to calculate a local mean value for the data set.

To conclude this section it shall be mentioned what Walter Smith, one of the two GMT developers, had to say concerning the well-known gridding algorithm *kriging*:

Strictly speaking, *kriging* is a particular interpolation technique, and GMT does not have anything that does exactly this. By experimenting with the option switches on **surface** or **nearneighbor** you may get a result that is just fine for what you had in mind, and may be close to what kriging would do.

For those who are interested:

kriging (named after a South African mining geologist) refers to interpolation by the following process:

1. Determine the autocovariance function of the data (the kriging literature refers to this in the somewhat transposed form of a „semi-variogram“).
2. Interpolate the data by a moving weighted average process, using the autocovariance to determine the weights so as to minimize the expected squared error in the interpolated estimate.

While this sounds good in theory and is *optimal* in the sense of minimizing the expected squared error, in practice there are some issues to contend with. First, the error minimization and the optimality can only be established for data having certain statistical properties (stationarity, ergodicity, and some restrictions on the form of the autocovariance function), and many datasets won't have these properties (stationarity, for example, so one has to remove a trend surface first and then do kriging on the residuals). Second, a practical algorithm cannot offer complete freedom in determining the autocovariance function empirically from the data; instead, algorithms support only one or a few functional forms for what kriging calls the „semi-variogram“, and one uses the data to fit parametric models to this and then uses this for the interpolation. This practical restriction means that the assumed form of the autocovariance actually employed by the routine is not the true autocovariance of the data, and this means that the optimality has been destroyed. So kriging is optimal in theory but maybe not in practice. Finally there is the problem of how you get a good estimate of a semi-variogram or a covariance function from data sparsely and irregularly spaced in the first place – these things are most easily calculated from gridded data, but if you had a grid you wouldn't be kriging in the first place. Concerns about these issues have stopped me from writing a kriging program for GMT so far. (This wasn't a problem for Krige in his original application; he was given samples (rock cores drilled) on an equidistant grid and his problem was to estimate the properties in the gaps between samples.)

In a certain sense, both **nearneighbor** and **surface** are also interpolating by moving weighted averages. This is obvious for **nearneighbor** and less obvious, though embedded in the finite difference equations, for **surface**. Thus both of these methods can give a result something like kriging; the only question is whether the choice of weighting scheme in these algorithms is close to what kriging would have chosen, or close to optimal. By playing with the tension parameter in **surface** you can change the weighting, which changes the power spectrum of the solution, which is equivalent to changing its autocovariance function.

A Colormaps

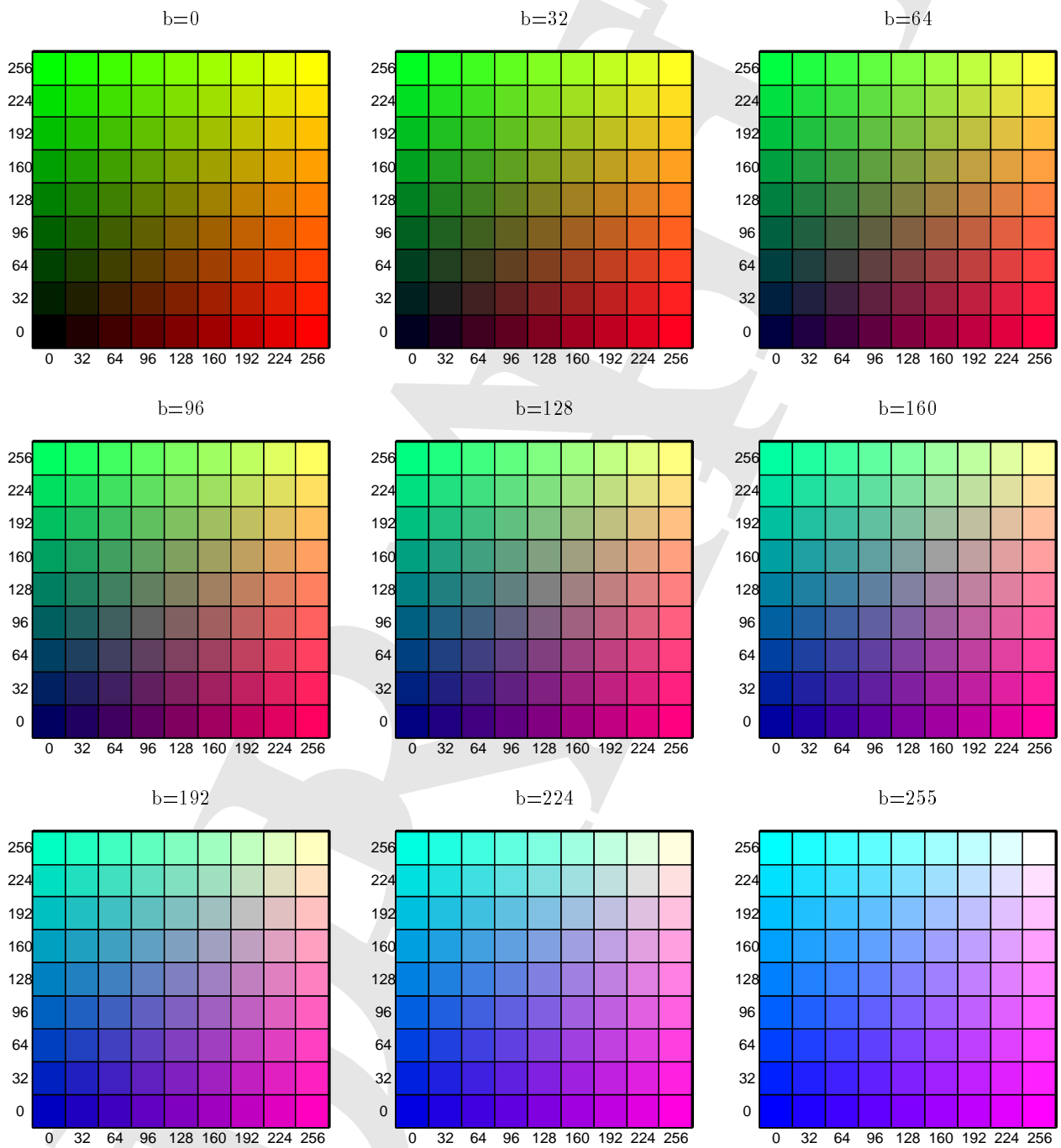


Figure A.15: The RGB color scheme

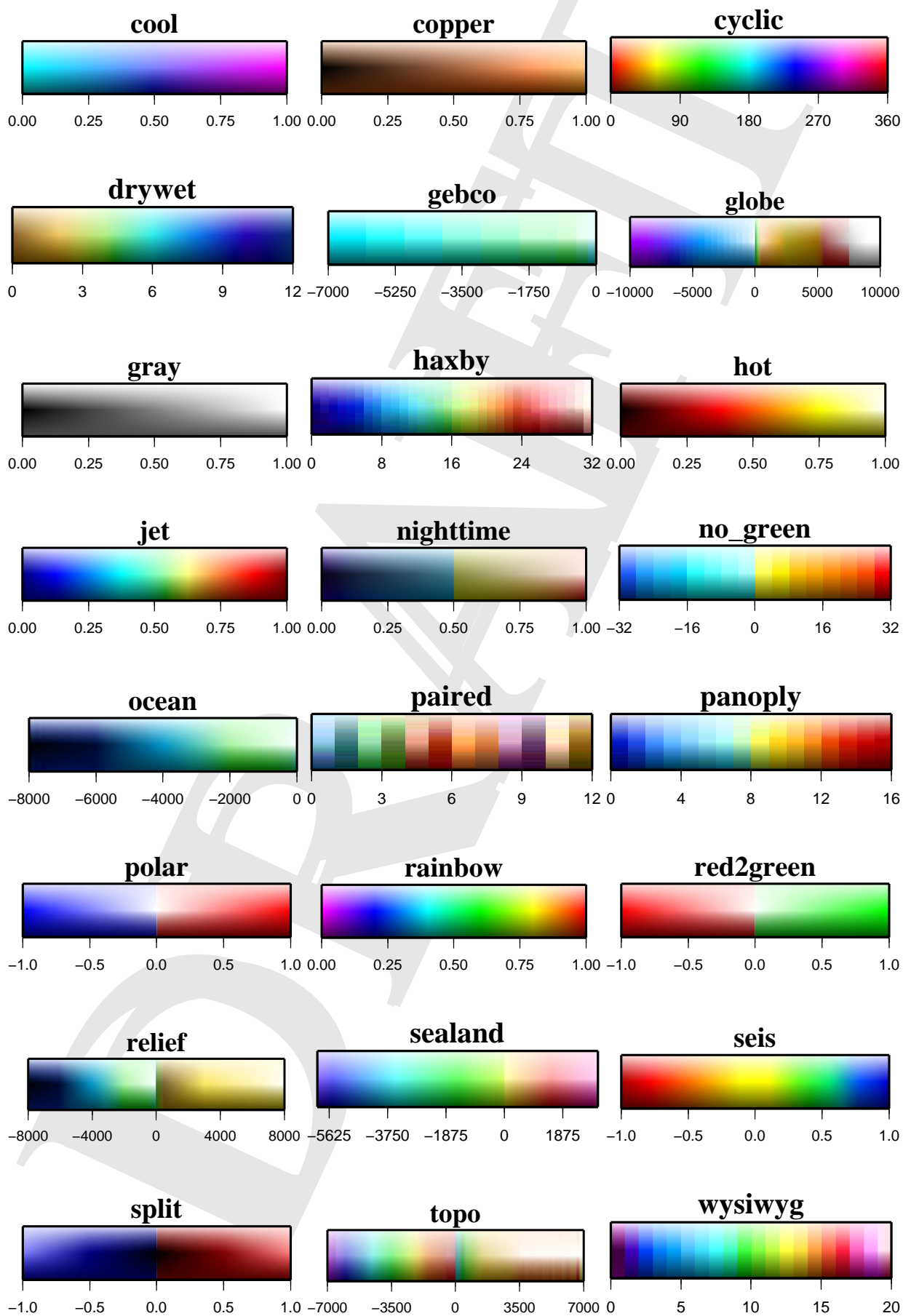


Figure A.16: The GMT CPT colormaps

B Useful tools

GMT can also be used for data analysis, some useful scripts can be found in this section.

B.1 Distance of two points on the earth surface

Script B.1 Calculation of the distance (in km) between two points with given geographical coordinates.

```

1  #!/bin/sh
2  LANG=C
3
4  usage="Usage: %s 'basename_$0' lon1 lat1 lon2 lat2"
5
6  if [ "$#" != "4" ] ; then
7      echo $usage
8      exit 1
9  fi
10
11 echo -n "Distance_(km): "
12 project -C$1/$2 -E$3/$4 -G1000 -Q | tail -1 | cut -f3

```

B.2 Tangent

Script B.2 Calculation of a tangent for a (x,y) dataset.

```

1  #!/bin/bash
2  # Linear Regression  $f(x)=a+bx$  from two-column data file  $(x,y)$ 
3
4  DATA=$1
5
6  XMIN='minmax -C $DATA | awk '{print $1}''
7  XMAX='minmax -C $DATA | awk '{print $2}''
8  #####
9
10 trend1d -Fxy $DATA -N2 -V > /dev/null 2> t
11 A0='grep "Polynomial" t | awk '{print $5}''
12 A1='grep "Polynomial" t | awk '{print $6}''
13
14 echo "f(x) = $A0 + $A1 * x"
15 rm t

```

B.3 Correlation coefficient

Script B.3 Calculation of the correlation coefficients (and the variances) of a (x, y) Dataset.

```

1  #!/ bin/ bash
2  LANG=C
3  #Diese Berechnung stimmt mit der in Fachlexikon Physik S.500
4  # angegebenen Formel ueberein.
5
6  DATA=$1
7
8  Variance_y='trend1d -N1 -Fr $DATA | awk '{ s += $1*$1; } END { print s/NR}''
9  Variance_r='trend1d -N2 -Fr $DATA | awk '{ s += $1*$1; } END { print s/NR}''
10
11 KK='calc "-p_config(\ "display\ ",2);_config(\ "leadzero\ ",1);\
12 _config(\ "tilde\ ",0);_sqrt(1-_$Variance_r/_$Variance_y)" | tail -n1 '
13
14 #echo "Variance_y = $Variance_y"
15 #echo "Variance_r = $Variance_r"
16 #echo "Korreklationskoeffizient = $KK"
17 echo $KK

```

Additional to `trend1d`, which was used in these examples, GMT provides many more tools to analyse and/or modify data, e.g. `trend2d`, `grdtrend`, `grdfft`, `grdfilter`, `fitcircle`, ...

C Sample Solutions

C.1 Solution to Exercise 5

Script C.1 Exercise's solution 5 (write_head_foot.sh)

```

1  #!/ bin/ bash
2  LANG=C
3  #####
4  function usage()
5  {
6      echo -e "\n__Usage: `basename $0` has to be called with\n\n\"
7          "____O<PostScript-File>____(output_file)\n\"
8          "____w<[K]O>____(write_header(K) or footer(O))\n\"
9          "____[-e]____(creates_eps, just valid for -wO)\n\"
10         "____[-f]____(force, overwrites PostScript-File)\n\"
11         "____[-s]____(show resulting postscript file)\n\"
12
13     exit 1
14 }
15 #####
16 function check_args()
17 {
18     OUT=NONE
19     HEADFOOT=NONE
20     FORCE=FALSE
21     while getopts efO:sw: OPT ; do
22         case $OPT in
23             O) OUT=$OPTARG ;;
24             e) EPSI=TRUE ;;
25             f) FORCE=TRUE ;;
26             s) SHOW_PS=TRUE ;;
27             w) HEADFOOT=$OPTARG ;;
28             *) usage ;;
29         esac
30     done
31     if [ $OUT == NONE ]; then usage
32     elif [ $HEADFOOT != K -a $HEADFOOT != O ] ; then usage
33     fi
34 }
35 #####
36 function write_head_foot()
37 {
38
39     if [ $HEADFOOT == K ]; then
40         if [ -e $OUT -a $FORCE == FALSE ]; then
41             echo -e "\nError: '$OUT' exists, use the -f option to overwrite\n"
42             usage
43             fi
44         psxy -R0/1/0/1 -JX1 -$HEADFOOT /dev/null > $OUT
45     else
46         psxy -R0/1/0/1 -JX1 -$HEADFOOT /dev/null >> $OUT
47
48         if [ "$EPSI" ]; then
49             ps2raster $OUT
50             rm $OUT
51             local OUT=${OUT%.ps}.eps
52             fi
53
54         echo "$OUT_created"
55         if [ $SHOW_PS ]; then gv $OUT; fi
56     fi
57 }
58 #####
59 check_args $*
60 write_head_foot
61 exit 0

```

C.2 Solution to Exercise 6

Script C.2 Exercise's solution 6 (bash_task.sh, Part 1)

```

1  #!/ bin/ bash
2  LANG=C
3
4  IN=./ data/ bash_ task. dat
5  TDAT=a5_ tmp. dat
6  TDAT2=a5_ tmp2. dat
7  OUT= bash_ task. ps
8  PRO= JX15/6
9  #####
10 function plot_ one()
11 {
12     local ANZREAL=$1
13     local YTXTPOS=$2
14     psxy "$ANN" $REG $PRO -Sc0.3 -G$COLOR $TDAT2 -K -O -Y$OFFSET >> $OUT
15     sort -n $TDAT2 | psxy $REG $PRO -W3/$COLOR -K -O >> $OUT
16     pstext -R0/10/0/10 $PRO -K -O -G$COLOR <<END >> $OUT
17     9.8 $YTXTPOS 13 0 1 MR $ANZREAL Sampling Size
18 END
19 }
20 #####
21 function select_ one()
22 {
23     local WHAT=$1
24
25     if [ $WHAT == M ]; then
26         local COLUMN=4
27         local ANNINC=0.01
28         local ANNTXT="Mean_ Value"
29         local OFFSET=8.5
30         local YINC=0.02
31     elif [ $WHAT == QM ]; then
32         local COLUMN=5
33         local ANNINC=0.01
34         local ANNTXT="Squared_ Mean_ Value"
35         local OFFSET=8.5
36         local YINC=0.02
37     elif [ $WHAT == V ]; then
38         local COLUMN=6
39         local ANNINC=0.0025
40         local ANNTXT="Variance"
41         local OFFSET=0
42         local YINC=0.005
43     else
44         echo "error_in_ 'select_ one()' "
45         exit
46     fi
47     local ANN="-B10f5 : Time_ Sampling_ Point : / a "$YINC" f "$ANNINC" : $ANNTXT : SWne"
48
49     grep -v '#' $IN | awk -v c=$COLUMN '{print $2,$c,$3}' > $TDAT
50     local REG='minmax -I10/$YINC $TDAT'

```

Script C.3 Exercise's solution 6 (bash_task.sh, Part 2)

```

51  for i in 500 2000 10000; do
52      if [ $i == 500 ]; then
53          local COLOR=200/0/0
54          local YTXTPOS=9
55      elif [ $i == 2000 ]; then
56          local COLOR=0/0/200
57          local YTXTPOS=8.2
58          local OFFSET=0
59      elif [ $i == 10000 ]; then
60          local COLOR=0/200/0
61          local YTXTPOS=7.4
62          local OFFSET=0
63      else
64          echo "error_in_`select_one()`"
65          exit
66      fi
67      awk -v w=$i '{if ($3=="Size="w) print $1,$2}' $TDAT > $TDAT2
68      plot_one $i $YTXTPOS
69  done
70 }
71 #####
72 function plot_all()
73 {
74     for i in V QM M; do
75         select_one $i
76     done
77 }
78 #####
79 ../tools/write_head_foot.sh -O$OUT -wK -f
80 if [ $? -ne 0 ]; then exit 1; fi
81 plot_all
82 ../tools/write_head_foot.sh -O$OUT -wO
83
84 gv $OUT
85 ps2raster -A -Te $OUT
86
87 rm $OUT $TDAT $TDAT2

```

C.3 Solution to Exercise 7

Script C.4 Exercise's solution 7 (land_coloured.sh)

```

1  #!/bin/bash
2  LANG=C
3
4  cp gmtdefaults4.base .gmtdefaults4
5
6  OUT=land_coloured.ps
7  IN=data/germany2pts.txt
8  INC=data/german_citys.txt
9  PRO=-JM15
10 REG=-R4/18/45/56
11 ANN=-B5/2WSEN
12 RP="$PRO_$REG_-K_-O"
13
14 gmtset CHAR_ENCODING = Standard+
15
16 function plot_karte()
17 {
18     gmtset BASEMAP_TYPE fancy
19     pscoast $RP "$ANN" -G200 -Na -Dh -A100 >> $OUT # Map with gray countries
20     psxy $IN $RP -m -G200/0/0 -L >> $OUT # Germany in red
21     gmtset LABEL_FONT_SIZE = 12 # Font size of the unit 'km'
22     gmtset LABEL_OFFSET = 0.1c
23 # Rivers, Sea, Borders, Scaling
24     gmtset HEADER_FONT_SIZE 14 HEADER_OFFSET 0.1
25     pscoast $RP -S32/155/128 -Lf6/55.5/52/200 k+1 -Tf6/54.2/1.7/3 -W0 -I1 -I2 -I8 -Dh -Na/5 -A100 >> $OUT
26     awk '{print $1, $2, 0.15+$4/5e6}' $INC |
27     psxy $RP -Sc -G0/0/200 >> $OUT # Cities as points
28     awk '{if($3=="Munich") print $1+.1, $2+.15, 1}' $INC |
29     psxy $RP -Skvolcano -G200/200/0 -W2/0 >> $OUT # Munich as volcano
30     awk '{if($3=="Munich") print $1, $2,6}' $INC |
31     psxy -L $RP -Gp300/8:F0/255/64B- -W2/0 -Skpentagon >> $OUT # Restricted area
32
33     awk '{print $1, $2-0.2, 12, 0, 1, "MC", $3}' $INC |
34     pstext $RP -G255 >> $OUT # Caption of the cities
35
36 # legend
37     gmtset ANNOT_FONT_SIZE = 12
38     pslegend -L1.0 $RP -Dx0/0/8.5/3.5/BL -F -G255 << END >> $OUT
39 H 14 1 Legend
40 D 0.5 2t10_10:0
41 N 2
42 V 0.3 2
43 S 0.3 c 0.2 0/0/200 0 0.6 cities
44 S 0.3 kvolcano 0.4 200/200/0 2 0.6 volcano
45 S 0.3 n 0.3 p300/8:F0/255/64B- 0 0.6 restricted area
46 S 0.3 g 0.3 200/0/0 0 0.6 Germany
47 V 0.3 2
48 N 1
49 D 0.5 2t10_10:0
50 G 0.2
51 M 10 52 300+1+jr f
52 G -0.1
53 L 12 33 MC GMT-Workshop
54 END
55 }
56
57 write_head_foot.sh -O$OUT -f -wK
58 plot_karte
59 write_head_foot.sh -O$OUT -s -wO -Te

```

C.4 Solution to Exercise 8

Script C.5 Exercise's solution 8 (projections_task.sh, Part 1)

```

1  #!/bin/bash
2  LANG=C
3
4  CITIES=cities.txt
5
6  function cities()
7  {
8      rm $CITIES 2> /dev/null
9      echo "-82.3833 23.1333_Habana" >> $CITIES
10     echo "139.7 35.6833_Tokyo" >> $CITIES
11     echo "72.850342 19.023174_Mumbai" >> $CITIES
12     echo "-87.654419 41.851151_Chicago" >> $CITIES
13 }
14
15 function proj()
16 {
17     local POPT='awk -v l1=$1 -v l2=$2 '{ if (l1==NR) { x1=$1; x2=$2 }
18         if (l2==NR) { y1=$1; y2=$2 }
19         } END { printf("-C%f/%f -E%f/%f", x1, x2, y1, y2) }' $CITIES'
20
21     local DIST=100 # distance of projected nodes in km
22     project $POPT -Q -G$DIST $3 # if $3 is set to -N a 'flat earth' is assumed
23 }
24
25
26 function plot()
27 {
28     local p=$1
29     local OUT=projection_.$p.ps
30     local REG=-R-180/180/0/90
31     local ANN=-Ba20g20/a20g20
32     local PRO
33
34     if [ $p == "Mercator" ]; then
35         ANN=-Ba60g20/a20g20SWne
36         REG=-R110/470/-60/80
37         PRO=-JM16
38     elif [ $p == "Gnomonic" ]; then
39         PRO=-JF0/90/72/14
40     elif [ $p == "Lambert" ]; then
41         PRO=-JA0/90/14
42     elif [ $p == "Stereographic" ]; then
43         PRO=-JS0/90/14
44     elif [ $p == "Hammer" ]; then
45         PRO=-JH16
46         REG=-R-180/180/-90/90
47     elif [ $p == "Gall-Peters" ]; then
48         PRO=-JY0/45/16
49         REG=-R-180/180/-90/90
50     else
51         echo "Unknown_Projection"
52         exit 1

```

Script C.6 Exercise's solution 8 (projections_task.sh, Part 2)

```

53 fi
54
55 local RP="-K-O_$REG_$PRO"
56
57 ../tools/write_head_foot.sh -wK -O$OUT -f
58
59 ## coastline
60 pscost $RP $ANN -DI -A15000 -G220 -S191/239/255 -W0.5p >> $OUT
61
62 psxy $CITIES $RP -W5/255/127/0 >> $OUT # great circle
63 psxy $CITIES $RP -W5/204/0/0 -A >> $OUT # loxodrome
64
65 # another way of plotting the great circle (between Habana&Tokyo)
66 proj 1 2 | psxy $RP -W5/50/200/255t10_30:0 >> $OUT
67 proj 2 3 | psxy $RP -W5/50/200/255t10_30:0 >> $OUT
68 proj 3 4 | psxy $RP -W5/50/200/255t10_30:0 >> $OUT
69
70 # another way of plotting the loxodrome (between Mumbai&Chicago)
71 proj 1 2 -N | psxy $RP -W5/0/0/200t10_30:0 >> $OUT
72 proj 2 3 -N | psxy $RP -W5/0/0/200t10_30:0 >> $OUT
73 proj 3 4 -N | psxy $RP -W5/0/0/200t10_30:0 >> $OUT
74
75 ## plot city points and annotate
76 psxy $CITIES $RP -Sc0.2 -G255/255/0 -W >> $OUT
77 awk '{print $1,$2,12,0,"Helvetica-Bold","TC", $3}' $CITIES |
78 pstext $RP -Dj0.2 -W255 >> $OUT
79
80 echo "L_14_1_BC_$p" > $$
81 echo "S_0.55_1_5/255/127/0_1.2_Great_Circle_(psxy)\n" >> $$
82 echo "S_0.55_1_5/204/0/0_1.2_Loxodrome_(psxy-A)" >> $$
83 echo "S_0.55_1_5/50/200/255t10_30:0_1.2_Spherical_Earth_Projection_(project)" >> $$
84 echo "S_0.55_1_5/0/0/200t10_30:0_1.2_Plane_Earth_Projection_(project-N)" >> $$
85 pslegend $RP -Dx0/0/8/2.7/BL -G200 -F $$ >> $OUT
86 rm $$
87
88
89 ../tools/write_head_foot.sh -wO -O$OUT
90 echo "$OUT_written"
91 gv $OUT
92 ps2raster -A -Te $OUT
93 }
94
95 cities
96 for p in Mercator Gnomonic Lambert Stereographic Hammer Gall-Peters; do
97     plot $p
98 done
99 rm $CITIES

```

C.5 Solution to Exercise 9

Script C.7 Exercise's solution 9 (nmea.sh)

```

1  #!/bin/bash
2  LANG=C
3
4  cp gmtdefaults4.base .gmtdefaults4
5
6  gmtset PLOT_DEGREE_FORMAT +ddd:mm
7
8  function get_data()
9  {
10     for ((i=1;i<=2;++i)); do
11         grep '^$GPGGA' data/nmea.day$i | awk -F, '{lon=substr($5,1,3);
12             lat=substr($3,1,2);
13             lon2=substr($5,4)/60.;
14             lat2=substr($3,3)/60.;
15             print lon+lon2,lat+lat2}' > $$.$i
16     done
17 }
18
19 function plot()
20 {
21     local OUT=nmea.ps
22
23     local COL1=200/0/0
24     local COL2=200/100/0
25
26     local REG1="-R7/9/53.3/54.22"
27     local PRO1="-JM15"
28
29     local W=7.85 E=7.95 S=54.13 N=54.2
30     local REG2="-R$W/$E/$S/$N"
31     local PRO2="-JM5"
32
33     local PCOAST="-K_-O_-G220_-S191/239/255_-W0.5p/0_"
34
35     ../tools/write_head_foot.sh -O$OUT -f -wK
36
37     pscoast $PCOAST $REG1 $PRO1 -Dh -Ba0.25g0.25/a0.25g0.25NEsw >> $OUT
38     psxy -R -J -K -O -L -W3/0/0/200<<END >> $OUT
39     $W $S
40     $W $N
41     $E $N
42     $E $S
43 END
44     psxy -R -J -K -O $$.$1 -W5/$COL1 >> $OUT
45     psxy -R -J -K -O $$.$2 -W5/$COL2 >> $OUT
46
47     gmtset BASEMAP_FRAME_RGB 0/0/200
48     pscoast $PCOAST $REG2 $PRO2 -Df -B10 -X1 -Y1 >> $OUT
49     psxy -R -J -K -O $$.$1 -W5/$COL1 >> $OUT
50     psxy -R -J -K -O $$.$2 -W5/$COL2 >> $OUT
51
52     ../tools/write_head_foot.sh -O$OUT -wO
53     ps2raster -A -Te $OUT
54     rm $OUT
55     gv 'echo $OUT | sed 's/.ps/.eps/g''
56 }
57
58 get_data
59 plot
60 rm $$.*

```

C.6 Solution to Exercise 12

Script C.8 Exercise's solution 12 (grdcontour_task.sh, Part 1)

```

1  #/bin/bash
2  LANG=C
3
4  int i = 7;
5  float f = 1.5;
6  string s = "hallo" ;
7
8
9
10 INGRD=./data/etopo5.grd
11 WORKGRD=work.grd
12 #####
13 function usage()
14 {
15     echo -e "\nUsage: `basename $0` _has to be called with_\n\n\"
16           \"_r<Region>_ ([A] lpen , [E] ropa; _default: _both)\n\"
17           \"_g<Darstellung>_ (grd [c] ontour , _grd [i] mage , _grd [v] iew ; _default: _c)\n\"
18     exit 1
19 }
20 #####
21 function check_args()
22 {
23     REGION="A_E"
24     HOW=c
25     while getopts g:r: OPT ; do
26         case $OPT in
27             r) REGION=$OPTARG ;;
28             g) HOW=$OPTARG ;;
29             *) usage ;;
30         esac
31     done
32 }
33 #####
34 function select_region()
35 {
36     WHAT=$1
37     OUT=contour_image_$WHAT_"_$SHOW.ps
38     if [ $WHAT == E ]; then # Europa
39         PRO=-JL10/43.5/35/50/15
40         REG=-R-10/30/35/59
41         ANN=-B10f5g5/5f5g2.5
42         MASTERCPT=relief
43     elif [ $WHAT == A ]; then # Alpen
44         PRO=-JC10/45.5/15
45         REG=-R5/15/43/48
46         ANN=-B1f1g1/1f1g1
47         MASTERCPT=sealand
48     else
49         echo "_error_in_'select'"
50         exit 1
51     fi
52
53     grdcut $INGRD -G$WORKGRD -R$REG
54 }
55 #####
56 function plot_ps()
57 {
58
59     if [ $SHOW == c ]; then
60         pscoast $REG $PRO $ANN -G200 -K > $OUT

```

Script C.9 Exercise's solution 12 (grdcontour_task.sh, Part 2)

```

61     grdcontour $REG $PRO $WORKGRD -O -C0.25 -A1/200/200/0 -G15/100 \
62         -Z0.001 -Nkm >> $OUT
63     elif [ $SHOW == i -o $SHOW == v ]; then
64         local CPT=color.cpt
65         local GRADGRD=gradient.grd
66     #     INGRDCUT=in_tmp.grd
67     #     grdcut $INGRD -G$INGRDCUT $REG
68
69     if [ $WHAT == A ]; then
70         local INGRDCUT2=in_tmp_2.grd
71         grdsample $WORKGRD -G$INGRDCUT2 -I1m
72         mv $INGRDCUT2 $WORKGRD
73     fi
74
75     makecpt -CMASTERcpt -Z > $CPT
76     grdgradient $WORKGRD -G$GRADGRD -Ne0.6 -A0/270
77     if [ $SHOW == i ]; then
78         grdimage $WORKGRD $REG $PRO -C$CPT -K -Y5 -I$GRADGRD > $OUT
79         pscoast $REG $PRO $ANN -W3 -O -K -A1000 -Di -I1 >> $OUT
80     else
81         grdview $WORKGRD $REG $PRO -C$CPT -K -Y5 -I$GRADGRD -JZ3 -E200/40 -Qi > $OUT
82     fi
83     psscale -O -C$CPT -I -E -D7.5/-1/15/0.5h -B2000:Topography:/:m: >> $OUT
84     rm $CPT $GRADGRD
85     else
86         echo "_error_in_'plot_ps'"
87         exit 1
88     fi
89     echo "'$OUT' created"
90     gv $OUT
91     ps2raster -A -Te $OUT
92     rm $OUT
93 }
94 #####
95
96 check_args $*
97
98 for i in $REGION; do
99     select_region $i
100     plot_ps
101 done
102 rm $WORKGRD

```

C.7 Solution to Exercise 14

Script C.10 Exercise's solution 14 (dem.sh, Part 1)

```

1  #!/bin/bash
2  LANG=C
3
4  function dem2grd()
5  {
6      for i in 1 2; do
7          if [ $i == 1 ]; then
8              local ZIN=./data/e020n90.tar.gz
9              local REG=-R20/60/40/90
10             GIN[$i]=./data/e020n90.grd
11         elif [ $i == 2 ]; then
12             local ZIN=./data/w020n90.tar.gz
13             local REG=-R-20/20/40/90
14             GIN[$i]=./data/w020n90.grd
15         else
16             echo ERROR
17             exit 1
18         fi
19
20         local IN='tar --wildcards -t "*DEM" -zf $ZIN'
21         if [ ! -e $IN ]; then
22             echo "extracting $IN from $ZIN"
23             tar --wildcards -x "*DEM" -zf $ZIN
24         fi
25         if [ ! -e ${GIN[$i]} ]; then
26             echo "gridding $IN to ${GIN[$i]}"
27             xyz2grd $REG $INC -G${GIN[$i]} -N-9999 -F -ZTLhw $IN
28             #grdinfo ${GIN[$i]}
29         fi
30     done
31 }
32
33 function get_region_of_interest()
34 {
35     if [ ! -e $AGRD ]; then
36         grdpaste ${GIN[1]} ${GIN[2]} -G$$
37         grdcut $$ $CUTREG -G$AGRD
38         rm $$
39         #grdinfo $AGRD
40     fi
41     if [ ! -e $ASGRD ]; then # resample for rougher resolution
42         grdsample $AGRD -G$ASGRD $LOWINC
43     fi
44 }
45
46
47 function plot_ps()
48 {
49     local MCOL=$1
50     local HOW=$2
51     local COL=$GMTHOME/share/cpt/GMT_$MCOL.cpt
52     local GRD=$AGRD
53     local PRO=-JA18.5/47/15
54     local REG=-R10/43/28/51r
55     local ANN=-B5g5/5g5
56
57     gmtset COLOR_NAN = 0/50/150
58     if [ $MASK == FALSE ]; then
59         local GIN=$GRD
60         local OUT=dem_$HOW$MCOL.ps

```

Script C.11 Exercise's solution 14 (dem.sh, Part 2)

```

61 else
62     local MASKDATA=./data/austria2pts.txt
63     grdmask $MASKDATA $INC $CUTREG -F -N0/1/1 -Gmask.grd
64     grdmath mask.grd $GRD MUL = t.grd
65     local GIN=t.grd
66     local OUT=dem_mask_SHOW$MCOL.ps
67 fi
68
69 grdgradient $GIN -Ggrad.grd -A0/270 -Ne0.6
70 if [ $SHOW == I ]; then
71     grdimage -C$COL $GIN $PRO $REG "$ANN" -Igrad.grd -K > $OUT
72     psscale -D16.5/5.1/9/0.5 -E -I -O -K -C$COL -B1000:Elevation:/:m:/ >> $OUT
73     echo "11.15_0_0_14_0_0_0_MC_$MCOL" |
74     pstext -JX15 -R0/10/0/10 -N -O >> $OUT
75 elif [ $SHOW == V ]; then
76     PROZ=JZ5
77     grdview $PROZ -C$COL $GRD $PRO $REG "$ANN" -Igrad.grd -K -E150/40 -Qc > $OUT
78     psscale -D4.5/1/9/0.5h -E -I -O -C$COL -B2000:Elevation:/:m:/ >> $OUT
79 else
80     echo "Error in `plot_ps()`_SHOW"
81     exit 1
82 fi
83
84 gv $OUT
85 # ps2raster -A -Te $OUT $OUT.epsi
86 ps2raster -A -Te $OUT
87 echo "created `echo $OUT | sed 's/.ps/.epsi/'`"
88 # eps2eps $OUT.epsi $OUTEPS
89 rm $OUT $OUT.epsi grad.grd t.grd 2> /dev/null
90 }
91
92
93 INC=10.5m
94 LOWINC=11
95 AGRD=./data/alp.grd
96 ASGRD=./data/alp_small.grd
97 CUTREG=R5/30/43/52
98
99 dem2grd
100 get_region_of_interest
101
102
103 #AGRD=$ASGRD; INC=$LOWINC # comment for high resolution
104
105 MASK=FALSE
106 for i in topo relief globe; do
107     plot_ps $i I
108 done
109 plot_ps relief V
110 plot_ps topo V
111 MASK=TRUE
112 plot_ps globe I
113 exit 0

```
