An introduction to Matlab: Data analysis, visualization and programming
August 16, 2001

Tore Furevik

Geophysical Institute, University of Bergen
and
Nansen Environmental and Remote Sensing Center

Translated by Ivar Seierstad, August 2001
Preface

The purpose of this document is to give the reader a quick introduction to Matlab; a powerful mathematical computing and visualization tool. More advanced users will hopefully also find some useful information.

After a brief introduction, we will in section 2 explain the Matlab Command Window. This includes search methods for functions, the help function and finally how to print and edit functions.

The cornerstone of Matlab is the matrix. In Matlab matrices may have one, two, three or even more dimensions. In section 3 we will explain the structure of the matrices and how to perform simple calculations, both ordinary matrix operations (linear algebra) and calculations element by element.

In section 4 we will learn how to save and retrieve files of different formats, both ascii and binary. Some of them have headers that we might want to ignore or extract information from.

In section 5 we will look at different ways of visualizing data. It will be shown that the graphics are object-oriented at different levels: the figure window, the axes and the graph level. Different plots in 2 and 3 dimensions will be demonstrated, together with ways of printing figures to file or printer.

We often need to display maps in Matlab. This is fortunately made simple by the plot-package called M_map, installed at the Geophysical Institute (and at the Nansen Center). In section 6 we will explain how to draw maps using this package, and how to use high resolution topography or coastline data for this purpose.

After working with Matlab for a while, one will discover just how useful the so-called M_files are. They consist of a collection of commands saved in a file with file extension .m. The properties of such files are explained in section 7. The M_files may be run either as a script or as a function. The former is equal to running the commands one by one at the command prompt.

After every section there are some exercises where you can practice the concepts in the preceding section. This will hopefully make the topics easier to grasp. For this purpose some datasets will be made available, although you may of course just as well use your own data. Well, all that remains is to wish you good luck with this document and your future Matlab career. And remember that if you encounter problems or want to share experiences, there is a user-group at the Geophysical Institute you can reach by sending email to matlab. And finally, never be afraid to ask someone!

Tore Furevik

Illustration on the front cover: A demonstration of a function adding arrows to figures.
Contents

1 Introduction .................................................. 4

2 The Command Window ........................................ 5
   2.1 Starting Matlab ........................................ 5
   2.2 Edit Commands ......................................... 5
   2.3 Matlab demos .......................................... 6
   2.4 The command lookfor: Is there a function that can solve your problem? .... 7
   2.5 The command help: How do you use a function? ........... 7
   2.6 The command type: What does the function look like? ....... 8
   2.7 The command format: How to change the print format on the screen? ....... 9
   2.8 The commands who and whos: What variables are in the Matlab workspace? 10
   2.9 Commands with !: UNIX or DOS commands from Matlab ........... 10
   2.10 Exercises to section 2 .................................. 11

3 Matrices in Matlab ............................................ 12
   3.1 To create a matrix ...................................... 12
   3.2 The semicolon operator: Suppressing output ............... 13
   3.3 The colon operator: Generating series .................... 13
   3.4 Matrix dimensions ..................................... 14
   3.5 Inf, -Inf og NaN: The use of non-numbers ................ 15
   3.6 Relation and logical operators .......................... 15
   3.7 Calculations element by element ........................ 16
   3.8 Matrix operations ...................................... 17
   3.9 Flow control: for, if and while ......................... 18
   3.10 Exercises to section 3 ................................. 19

4 To read and write data ...................................... 21
   4.1 Reading ascii data without header ....................... 21
      4.1.1 Reading CTD data .................................. 21
      4.1.2 Reading gridded topography data .................... 22
   4.2 Reading ascii-data with header .......................... 23
      4.2.1 Reading file with uninteresting header ............... 24
      4.2.2 Reading file with interesting header ................. 24
   4.3 Reading binary files ................................... 25
      4.3.1 Reading simple binary data .......................... 25
      4.3.2 Reading unformatted binary data saved in Fortran .... 26
   4.4 Saving ascii-data ....................................... 28
   4.5 Saving binary data ..................................... 29
   4.6 Exercises to section 4 .................................. 29
5 Graphics
  5.1 Object-oriented graphics ........................................ 31
  5.1.1 The figure level ........................................... 31
  5.1.2 The axes level ............................................ 33
  5.1.3 The graph level ........................................... 34
  5.1.4 Adding text ................................................ 35
  5.2 The colors in Matlab ........................................... 37
  5.3 Two-dimensional plots .......................................... 38
    5.3.1 Reading the data to be plotted .......................... 38
    5.3.2 The time variable ....................................... 40
    5.3.3 The function plot ....................................... 40
    5.3.4 Logarithmic plots ...................................... 43
    5.3.5 Histogram and bar chart ................................ 43
    5.3.6 Plots in polar coordinates ............................... 45
    5.3.7 Stick-plot .............................................. 46
    5.3.8 Plotting two variables on the same graph .............. 47
  5.4 Three-dimensional plots ....................................... 48
    5.4.1 The functions contour and contourf ..................... 49
    5.4.2 The other 3D plot functions ............................. 50
  5.5 Printing figures ............................................... 52
  5.6 Exercises to section 5 ........................................ 53

6 Map plotting ..................................................... 54
  6.1 Choice of projection .......................................... 54
  6.2 Plotting the map ............................................. 55
  6.3 Plot data or add text to a map ................................ 55
  6.4 Coastal data ................................................ 57
  6.5 Topography ................................................... 58
  6.6 Exercises to section 6 ....................................... 60

7 Scripts and functions: Create your own Matlab routines .... 62
  7.1 Matlab scripts ............................................... 62
  7.2 Matlab functions ............................................ 65
  7.3 Exercises to section 7 ....................................... 68
Figure 1: Graphics produced with Matlab: Temperature and ice cover simulated with the 'Bergen Climate Model', a coupled atmosphere-ocean model

1 Introduction

The name Matlab is short for Matrix Laboratory, indicating that matrices are the building blocks of Matlab. The matrices or arrays, as we also call them, can be of all dimensions. From simple elements (1x1 matrices) to vectors (1xN matrices) and multiple dimensional matrices (MxNx...) A useful feature in Matlab is that the matrix dimensions need not be declared in advance, as opposed to other programming languages such as Fortran and Pascal. The time you will spend to perform complex calculations and produce high quality graphics in Matlab is usually far less than Fortran, C, Basic, Pascal or other languages. In that sense Matlab is outstanding. Matlab has a lot of built-in functions together with hundreds of other functions collected in so-called toolboxes. One of these is the statistical toolbox containing functions for calculating correlations, mean values, variances and approx. 200 other things. In this reference we can obviously only describe a small fraction of the enormous number of functions and commands available in Matlab.
2 The Command Window

In this section we will explain how to start Matlab, together with basic functions and commands such as demo, lookfor, help, type, whos and !.

2.1 Starting Matlab

To start Matlab on a Unix terminal, just type Matlab and press Enter. Remember that not all computers have Matlab installed. At the Geophysical Institute Matlab is installed on the servers Storm and Virvel. At the Nansen Institute it is installed on the server Troll.

If you work on another computer or a workstation, remember to set the screen display to your computer, otherwise you will not be able to see the graphics. The number of the computer is usually written on the case, either as text or as a number. ie 129.177.63.138 is the number of the computer tyfon.gfi.uib.no. So before you start Matlab, type setenv DISPLAY your-computer-number or setenv DISPLAY your-computer-name.

If you work on a PC or a Mac you can probably start Matlab simply by double-clicking the Matlab icon. Assuming obviously that you have Matlab installed on your computer.

2.2 Edit Commands

If all went well, you will now see the Matlab prompt \(\gg\) on your screen. Matlab is now ready for your commands. We start by writing something rubbish,

\[
\gg \text{hsgdh}
\]

and Matlab replies with

\textbf{??? Undefined function or variable 'hsgdh'}

Matlab obviously didn’t understand our command.

Let us try something more sensible:

\[
\gg \sin(1)
\]

Matlab now replies with

\[
\text{ans} =
0.8415
\]

and we have managed to calculate the sin of the angle 1 in radians.

Now it is time to learn about the arrow keys on your keyboard. They are useful for flicking through your recent commands. The up-arrow (\(\uparrow\)) displays the command \(\sin(1)\), thus repeating the last command. If you use it once more, \texttt{hsgdh} appear at the command prompt. However, if you use the down-arrow (\(\downarrow\)) the command \(\sin(1)\) will reappear. This was our last command so another down-arrow will have no effect. You can edit the command by pressing the left-arrow (\(\leftarrow\)) once and type \(\theta\) instead of \(1\). Press Enter,

\[
\gg \sin(2)
\]

\[
\text{ans} =
0.9093
\]

and we have found the sin of 2. Note that Matlab is case sensitive so that
Furevik: An introduction to Matlab

(a) \hspace{5cm} (b)

Figure 2: Cruller (a) and Hoops (b) figures in Matlab’s demo section 2.3.

\[ \text{SIN}(2) \]
will return the error

?? Undefined function or variable ‘SIN’.

The right-arrow (→) moves the marker one letter to the right. A nifty feature is that if you type a few letters followed by the up-arrow, the last command beginning with these letters will appear at the command prompt. Try \( \ll \text{h} \rightarrow \). If you want to quit Matlab you can do this with

\[ \text{quit} \]
and pressing Enter.

2.3 Matlab demos

After this somewhat boring start we can hopefully regain some inspiration by running some of Matlab’s demos. Some of Matlab’s features are well demonstrated if you type demo at the command prompt. A new window will appear on the screen (assuming that the Unix command setenv is correctly used, see section 2.1), and from the left menu you can choose among different subjects. We can try Gallery as an example.

Two of the figures demonstrated here are shown in figure 2. They are the Cruller and Hoops figures. Everyone should spend some time exploring this demo in order to get an impression of some of the opportunities on offer. However, note that making such nifty graphics is not straightforward! You can look at other interesting demos by typing tour. The solution to the travelling salesman problem (visiting a number of points with the shortest possible distance), for instance, can be found by typing travel.
2.4 The command **lookfor**: Is there a function that can solve your problem?

The *lookfor* command is very useful. It searches through all Matlab functions and enables us to find out whether Matlab has a function for our purpose. Suppose we want to find the mean value of some numbers and want to find out if Matlab has such a function. We try with

```matlab
>> lookfor mean
```

and after a lot of work Matlab will display a list of about 40 functions containing the word *mean*. The first five are

- **MEAN** Average or mean value. GWNOISE generate valid mean value, standard deviation and seeds for GWNOISE block.
- **FRONTCON** Mean-variance efficient frontier with portfolio constraints EXCELPORTOPT Calculate the mean-variance efficient frontier. MEAN2 Compute mean of matrix elements.

and it should be obvious that the first function is what we need.

Even though the *lookfor*-function is slow, it only searches through the first line of text in all the functions. If we want to search through the entire function, we will have to write

```matlab
>> lookfor mean -all
```

This will display a lot more functions. If you already have an idea of what you are looking for, you may try with

```matlab
>> which mean,
```

that will display

```matlab
/usr/matlab531/toolbox/matlab/datafun/mean.m
```

and shows that there is a function named *mean.m* in the mentioned path.

2.5 The command **help**: How do you use a function?

In the section above we found the command we needed, namely the *mean*. But still we have no clue of how to use this command. In such cases we use the helpful *help* command. Try to type

```matlab
>> help mean,
```

and the following text is displayed

**MEAN** Average or mean value.

For vectors, **MEAN(X)** is the mean value of the elements in X. For matrices, **MEAN(X)** is a row vector containing the mean value of each column. For N-D arrays, **MEAN(X)** is the mean value of the elements along the first non-singleton dimension of X.

**MEAN(X,DIM)** takes the mean along the dimension **DIM** of X.

**Example**: If $X = [0 \ 1 \ 2$

$$
3 \ 4 \ 5
$$

8
then mean(X,1) is [1.5 2.5 3.5] and mean(X,2) is [1 4]

See also MEDIAN, STD, MIN, MAX, COV.

The help texts in Matlab are informative and nearly always consist of useful examples. In this case there is an example with a 2D matrix X, which you can create with

\[ X=[0 1;3 4] \]

How to create matrices will be explained in more detail in section 3. If we now write

\[ \text{mean}(X), \]

we get the answer

\[
\text{ans} = \\
\begin{bmatrix}
1.5000 & 2.5000 & 3.5000
\end{bmatrix}
\]

and we can see that Matlab returns the mean value for each of the three columns. This is equivalent to \( \text{mean}(X,1) \). That is, we calculate the mean value of the first dimension (the columns). If we want the mean value of every row, we will need to type

\[ \text{mean}(X,2), \]

and Matlab answers with

\[
\text{ans} = \\
\begin{bmatrix}
1 \\
4.
\end{bmatrix}
\]

Note that in the last line of the \texttt{mean} help text there are references to some associated functions: median, \texttt{std} (standard deviation), \texttt{min,max} (minimum and maximum value of a matrix) and \texttt{cov}(covariance).

2.6 The command \texttt{type}: What does the function look like?

All functions not built into the Matlab-system can be displayed with the \texttt{type} command. As an example,

\[ \text{type} \text{ mean} \]

will display the following text

\[
\text{function } y = \text{ mean}(x,\text{dim}) \\
\%	ext{MEAN} \quad \text{Average or mean value.} \\
\% \quad \text{For vectors, } \text{MEAN}(X) \text{ is the mean value of the elements in } X. \text{ For} \\
\% \quad \text{matrices, } \text{MEAN}(X) \text{ is a row vector containing the mean value of} \\
\% \quad \text{each column. For N-D arrays, } \text{MEAN}(X) \text{ is the mean value of the} \\
\% \quad \text{elements along the first non-singleton dimension of } X. \\
\%
\%
\% \quad \text{MEAN}(X,\text{DIM}) \text{ takes the mean along the dimension DIM of } X. \\
\%
\%
\% \quad \text{Example: If } X = [0 1 2 \]
\%
\% \quad \begin{bmatrix}
3 & 4 & 5
\end{bmatrix}
\]

9
This shows the structure of a Matlab function. We will return to this in section section 7.2. The function also contains some other Matlab functions \texttt{(nargin, dim, min, find, size, if, isempty, end, sum, else)}, which we will go through in the same section.

The sign \% is a comment-mark which is ignored when Matlab runs a script. Comment-lines are very important for giving information about a function both to your self and others. When we used the \texttt{lookfor} command in section 2.4 it displayed the first comment-line. This can therefore be said to be the headline of the function. The \texttt{help} command (see section 2.5) displays the longest connected comment-lines. So that the lines with Copyright and Revision are not displayed. Bear this in mind when you build your own functions so that the most important information is written as comments on the first few lines, with details about the script further down.

2.7 The command \texttt{format}: How to change the print format on the screen?

The default format of Matlab is as shown when we issued the command \texttt{sin(1)} above. That is with 4 figures after comma. This can be altered with the \texttt{format} command. Try writing

\begin{verbatim}
>> format long
>> sin(1)
\end{verbatim}

and note the difference. Now we have 14 figures after comma. The command \texttt{format short} will return the format to the default value. Type \texttt{help format} to view the range of formats available in Matlab.
2.8 The commands \texttt{who} and \texttt{whos}: What variables are in the Matlab workspace?

When you work in Matlab all variables are stored in memory. After a while it will certainly be difficult to remember the name and size of all of them. Then the \texttt{who} and \texttt{whos} commands are very useful. The former lists only the name of the variables, while the latter lists name, size of matrices, format and the size of memory they occupy. You may try

\begin{verbatim}
>> who
\end{verbatim}

and Matlab will display something like

\begin{table}[h]
\begin{tabular}{llll}
\hline
Name & Size & Bytes & Class \\
\hline
X & 2x3 & 48 & double array \\
an & 2x1 & 16 & double array \\
\hline
\end{tabular}
\end{table}

\texttt{Name} \begin{tabular}{l}
\texttt{Grand total is 8 elements using 64 bytes}
\end{tabular}

These are the variables currently in the Matlab memory. The first, \texttt{X}, has dimensions 2x3 (2 rows, 3 columns), use 48 bytes (8 bytes pr figure) and a table of numbers with double precision. The \texttt{ans} variable is the result of the last calculation. In this case the answer to \texttt{mean(X,2)}. If you write

\begin{verbatim}
>> ans
\end{verbatim}

you will see that the variable \texttt{ans} is a 2x1 matrix (2 rows, 1 column) occupying 16 bytes of memory since Matlab operates with double precision. In total we have 8 elements (2x3+2x1) using 64 bytes of memory. If we only want to see the variables beginning with a specific letter, we can write

\begin{verbatim}
>> whos X*
\end{verbatim}

and only the one variable is listed.

Variables can be removed from memory by using the \texttt{clear} command. Type \texttt{clear X} and then \texttt{whos}. You will now discover that the variable \texttt{X} no longer resides in the Matlab memory.

2.9 Commands with \texttt{!}: UNIX or DOS commands from Matlab

Another useful feature in Matlab is the ability to run UNIX or DOS commands from Matlab. You can do this by starting your commands with an exclamation mark. Try typing \texttt{ls -al} or \texttt{dir}. All operating-system commands can be performed from the Matlab window. If you work in UNIX, try

\begin{verbatim}
>> !vi /usr/matlab531/toolbox/matlab/datafun/mean.m,
\end{verbatim}

This will give you the opportunity to edit the \texttt{mean} function. This is a clever thing to do when building your own functions and want to "steal" a good beginning from another function. Just save the file with your own filename and edit it to your liking (I renounce any responsibility for copyright misuse!).
2.10 Exercises to section 2

In this section we have become familiar with the Matlab Command Window and how to type commands in Matlab. Here are some exercises to explore this topic further:

1. Start Matlab either on a PC or UNIX terminal, and run some of Matlabs demos such as tour and demo.

2. Type the following matrix: \( X=[1 \ 0 \ 4 \ 9 \ 2; \ 3 \ 9 \ 4 \ 2 \ 5; \ 3 \ 4 \ 2 \ 6 \ 4; \ 2 \ 3 \ 4 \ 8 \ 3; \ 2 \ 9 \ 3 \ 2 \ 1] \).
   
   (a) What is the mean value of each column and row?
   
   (b) What is the minimum value of each column and row (hint: Use the functions lookfor and help)?

   (c) What is the mean value, minimum value, standard value and variance of the matrix (all numbers)?

   (d) Type mean=2, and try to calculate the mean value again. What does the error message say? What can you do to avoid this error (hint: you don't need to restart Matlab)?

3. In Matlab there are several predefined constants in the same way as there are predefined functions. Some of these are \( \pi \), \( i \) and \( j \). The first being the ratio between the circumference and diameter of a circle, and the two latter ones are both the imaginary unit vector.

   - What is the area and volume of a sphere with radius 6371 km?
   - What is \( \exp(i*\pi) \)? What do you get if you add 1? Why?
   - Calculate \( (1+i)^4 \) and \( \sin(2)+i*\sinh(2i) \).
3 Matrices in Matlab

In section 2 we learned about the Matlab Command Window and some commands that make life as a Matlab user easier. In this section we will explain the use of matrices in Matlab. Even though the use of graphics will be covered in section 5, we will introduce some plotting routines to illustrate what we are doing.

3.1 To create a matrix

It is easy to create matrices in Matlab. The simplest matrix (1x1 matrix) can be created by typing

\[ A=3.14 \]

Some variables are predefined; try for instance to write \( pi \), \( i \), or \( j \) at the command prompt. The last two are the unit vectors along the imaginary axis (complex numbers). But all variables (and functions) can be used to other things if you want to. We can type vectors as a series of numbers in square brackets,

\[ B=[1 \ 9 \ 3 \ 5 \ 2 \ 4 \ 6 \ 8 \ 3 \ 9] \]

which constructs the variable \( B \) as a 1x10 matrix. In order to make a more general matrix each row will have to be separated by a semicolon. For example:

\[ C=[1 \ 2 \ 3 \ 4; \ 3 \ 4 \ 5 \ 6; \ 8 \ 6 \ 4 \ 2] \]

This creates a 3x4 matrix. Note that the number of elements must be the same in every row or every column.

If you want to extract elements from a matrix, you can do this by giving the row and column number of the matrix. So that

\[ C(1,1) \]

gives \textbf{ans} =

\[ 1 \]

and

\[ C(2,3) \]

gives \textbf{ans} =

\[ 5 \]

Note that Matlab will interpret a single index number as a row number. If the index number is larger than the number of rows, it will start over in the next column. So that \( C(5) \) becomes 4 (in the second column) and \( C(8) \) becomes 5 (third column). For vectors you obviously use only one index number so that \( B(3) \) returns the value 3.

It is also useful to know some important functions that enable us to produce matrices with random numbers. These functions are \texttt{zeros}, \texttt{ones}, and \texttt{rand}. As the names indicate, they produce zero-matrices, one-matrices and random-matrices respectively. Some examples are:

\[ \texttt{zeros}(2,3) \]
\[ \texttt{ones}(2,4) \]
\[ \texttt{rand}(6,2) \]
Note that if we only use one number inside the parenthesis, the functions will produce a square matrix with this number of rows and columns. And then a not so useful command, but included here as a curiosity, is the function \texttt{magic(N)}. It forms magical squares where the sum of each row and column together with the diagonals are constant. While at the same time using all numbers from 1 to NxN. Try yourself with \texttt{magic(5)} where all the sums equal 65, or with \texttt{magic(10)} where all the sums equal 505.

3.2 The semicolon operator: Suppressing output

Note that for all the commands above the values were displayed on the screen. This can sometimes be useful, but we often work with large variables and then such printing can take anything from minutes to hours. To convince yourself, try typing \texttt{magic(1000)} (my computer used approx. 5 minutes). To avoid this, just remember to end the line with a semicolon (;). Try typing

\begin{verbatim}
\texttt{A=3.14;}
\end{verbatim}

and note the difference to the command given in section 3.1.

A good advice is to always end your commands with a semicolon, unless you specifically need to see the value on the screen. The semicolon also enables us to write more Matlab commands on the same line. This sometimes leads to more orderly programming.

3.3 The colon operator: Generating series

An important concept for use in matrices is the colon operator (:). We use it for generating series. A simple example is:

\begin{verbatim}
\texttt{a=1:10}
\end{verbatim}

which generates all integers from 1 to 10. You can adjust the step length by including a number in the middle. This number may be positive or negative, integers or fractions. ie

\begin{verbatim}
\texttt{a=1:3:10;}
\texttt{b=1:-2:-8;}
\texttt{c=0:.5:2;}
\texttt{d=0:pi/6:2*pi;}
\end{verbatim}

The colon operator is very flexible so that the matrix \texttt{C} in section 3.1 can be created with

\begin{verbatim}
\texttt{C=[1:4;3:6;8:-2:2];}
\end{verbatim}

For large matrices this is very efficient. The colon operator can also in a similar way be used to index matrices, so that

\begin{verbatim}
\texttt{C(1:2,3)}
\end{verbatim}

extract the first two elements in the third column (3 and 5),

\begin{verbatim}
\texttt{C(3,2:4)}
\end{verbatim}

extract the three last elements of the third row (6, 4 and 2), and

\begin{verbatim}
\texttt{C(:,3:4)}
\end{verbatim}

gives the two last columns.

You can change the sequence of a matrix by typing:
\[ C = C(:,[2 \\ 1 \\ 3 \\ 4]); \]
which interchanges the first and second column. If you want to remove the third column, you may do this with
\[ C1 = C(:,[1:2 \\ 4]); \]
and two matrices can be joined together with
\[ C2 = [C \\ C \\ C1; C1 \\ C1]; \]
producing the matrix
\[
\begin{array}{ccccccccc}
2 & 1 & 3 & 4 & 2 & 1 & 3 & 4 & 2 \\
4 & 3 & 5 & 6 & 4 & 3 & 5 & 6 & 4 \\
6 & 8 & 4 & 2 & 6 & 8 & 4 & 2 & 6 \\
2 & 1 & 4 & 2 & 1 & 3 & 4 & 2 & 1 \\
4 & 3 & 6 & 4 & 3 & 5 & 6 & 4 & 3 \\
6 & 8 & 2 & 6 & 8 & 4 & 2 & 6 & 8 \\
\end{array}
\]

We often use the end command together with the colon operator. By typing \( C(3:end) \) we will get all elements from the third to the last. This is particularly useful when the length of a variable is unknown, as is often the case when we retrieve data from a file (see section 4.2).

### 3.4 Matrix dimensions

In order to exploit the matrix properties of Matlab, we need to be able to manipulate matrix dimensions. Here is an example where we start out with a 4-dimensional random matrix:
\[ x = \text{rand}(2,3,7,5); \]
The matrix \( x \) has 2, 3, 7 and 5 elements along the four 'directions', and consists of a total of 210 elements. The total is the product of the number of elements for every direction, \( \text{prod} (\text{size}(x)) \). To find the size of the matrix, just type \( \text{size}(x) \) returning the vector \([2 \ 3 \ 7 \ 5]\). The length of the matrix, the maximum number of elements in one direction, can be found with \( \text{length}(x) \) or \( \text{max}(\text{size}(x)) \).

All numbers in the matrix \( x \) are stored as a sequence in memory, although some additional information tells Matlab that this sequence is a 4D matrix. If we for instance write \( x(1:7) \), you will get the first 7 elements of the sequence, that is \( x(:,1,1,1), x(:,2,1,1), x(:,3,1,1) \) and \( x(1,1,2,1) \). A clever function that can change the Matrix dimension, without altering the sequence of the numbers, is \( \text{reshape} \). Try typing
\[ y = \text{reshape}(x,2,3,35); \text{size}(y) \]
\[ z = \text{reshape}(x,\text{prod}(\text{size}(x)),1); \text{size}(z) \]
Note that the \( \text{reshape} \) command will only change the information about how the dimensions should be, not the number of elements in the matrix. This is useful when we have a dataset consisting of values for every month for some years, and we want to find the mean value for January. Note also that the \( \text{reshape} \) command never changes the data sequence stored in memory. So that \( \text{reshape}(y, \text{size}(x)) \) followed by \( \text{reshape}(z, \text{size}(x)) \) results in the original \( x \). The function
ind2sub can be used to find a position in the matrix. Try writing

\[
\begin{align*}
\text{>> } & [\text{val}, \text{ind}] = \text{max}(x(:)) \\
\text{>> } & [a, b, c, d] = \text{ind2sub}(\text{ind}, \text{size}(x))
\end{align*}
\]

On the first line we find the maximum value of the matrix (slightly less than one) and its position in the sequence (a number between 1 and 210). This position may not be very helpful since it is usually more convenient to know the position in the matrix, which can be found with \text{ind2sub}. We then find the maximum value in \( x(a, b, c, d) \). Note that we need four indices since we have a 4D matrix.

### 3.5 Inf, -Inf og NaN: The use of non-numbers

Matlab does not break down when it encounters 'illegal' operations such as dividing by zero. This element, in such cases, will be assigned the value infinity (\text{inf}), minus infinity (\text{neg} \text{inf}) or a non-number (\text{NaN}). We can produce these three cases by typing 1/0, -1/0 or 0/0 respectively. Note that Matlab always displays this message: \text{Warning: Divide by zero}. In further calculations with such elements the result will still be undefined. Try typing

\[
\text{>> } \text{sum([1 NaN 2])};
\]

### 3.6 Relation and logical operators

Matlab supports the use of relational and logical operators. The purpose of such operators is to provide answers to True-False questions. Relation operators may be: equal (==), not equal to (\neq), less than (<), greater than (>), less than or equal to (\leq), greater than or equal to (\geq), and (\&), or (|), and not (.). Boolean functions, returning the value true or false, can be: \text{find}, \text{isempty}, \text{isfinite}, \text{isnan}, \text{isnumeric}, \text{isreal}, \text{isprime} og \text{ischar}. The output of all relational or logical expressions is 1 for True and 0 for False.

To illustrate how to use some of them, we first create a vector \( x \)

\[
\text{>> } x=[1:5:30 \text{ NaN } 20:-2:4 \text{ Inf}]
\]

This is a vector with 17 columns. So if we type

\[
\text{>> } I=\text{isfinite}(x)
\]

it returns a vector with 1 where the elements of \( x \) are finite numbers, and 0 where they are not. We proceed by finding the indices to all elements of value 1 in \( I \). For this purpose we always use the function \text{find}: 

\[
\text{>> } J=\text{find}(I)
\]

and we can see that elements number 7 and 17 are omitted. If we want to remove these elements from the vector \( x \), just type

\[
\text{>> } x=x(J)
\]

The vector \( x \) should now consist of only finite numbers. The elements \text{NaN} and \text{Inf} are omitted. However, this was cumbersome. The three operations above can be done in a single expression

\[
\text{>> } x=x(\text{isfinite}(x));
\]

If we want to replace all numbers greater than 15 with zeros, we can in the same way type
Figure 3: (a) The functions \( \sin(x) \) and \( \cos(x) \). (b) The functions \( \sqrt{x} \), \( x^{1.5} \), and \( x^2 \). The commands generating these vectors are shown in section 3.7.

\[
\gg x(\text{find}(x>15))=0
\]
and if we for some reason want to square all prime numbers, we can type
\[
\gg x(\text{isprime}(x))=x(\text{isprime}(x)) \cdot 2
\]
As you can see, there were only 11 prime numbers. Finally we place all numbers greater than or equal to 10 in a new vector \( y \),
\[
\gg y=x(\text{find}(x>=10))
\]
The functions \text{isnumeric} and \text{ischar} checks whether a variable consist of numbers or characters. If \( A=’hei’ \) and \( B=100 \), the \text{isnumeric}(A) function will return the value 0, while \text{ischar}(A) returns 1. For \( B \) it will be the other way round.

### 3.7 Calculations element by element

Now that you know how to create matrices the next step is to perform simple calculations with them. We will start with an example where we first generate a sequence of numbers from 0 to 10 with step length 0.01. Then we calculate the sinus and cosine to all the numbers and visualize them with the \text{plot} function:
\[
\gg x=0:.01:10;
\]
\[
\gg y1=\sin(x);
\]
\[
\gg y2=\cos(x);
\]
\[
\gg \text{plot}(x,y1,’-’,x,y2,’-‘);
\]
The graphics in Matlab will be covered in more detail in section 5. We plot \( y1 \) as a function of \( x \) with a solid line, and \( y2 \) as a function of \( x \) with a dashed line. The results are shown in
figure 3a.
We will now try to calculate the square root of the sequence \( x \). This can be done with the Matlab function \( \text{sqrt} \) or directly with \( x^{0.5} \).

An attempt with

\[
\gg y1 = x^{0.5}
\]

will yield the error message:

??? Error using ==> ^
Matrix must be square.

This is because Matlab thinks you are performing a matrix operation. If you want to perform operations such as multiplication or division element by element, you will have to include a dot before the operator. That is, to calculate the square root of \( x \)

\[
\gg y1 = x^{0.5};
\]

and \( x^{1.5} \) and \( x^2 \) can be done with

\[
\gg y2 = x^{1.5};
\]

\[
\gg y3 = x^2;
\]

\[
\gg \text{plot}(x,y1,':',x,y2,':',x,y3,':');
\]

The plot is shown in figure 3b. The difference between matrix operations and calculations element by element, can be shown by using the matrix \( C1 \) from section 3.3 (\( C1=[2 \ 1 \ 4; 4 \ 3 \ 6; 6 \ 8 \ 2] \)) and the two seemingly similar commands

\[
\gg C1*C1
\]

\[
\gg C1.*C1
\]

The calculations above can also be performed on matrices with 2 or more dimensions. A useful tool for such operations is the function \( \text{meshgrid} \). This function spans a grid where points in a 2D matrix are assigned \( X \) and \( Y \) positions. (For 3D matrices \( X, Y, \) and \( Z \) positions). Assume, for instance, that we we want to calculate the cosine to \( X \times Y \) on an area limited by \( X = \pm 3 \), and \( Y = \pm 3 \). We first make the grid

\[
\gg [X,Y]=\text{meshgrid}(-3:.1:3,-3:.1:3);
\]

creating two matrices of size 61x61. You will then get the plots shown in figure 4 by typing

\[
\gg \text{figure}(1); Z=X.*Y; \text{surf}(X,Y,Z);
\]

\[
\gg \text{figure}(2); Z=\cos(X.*Y); \text{surf}(X,Y,Z);
\]

We will return to the plot function \( \text{surf} \) in section 5. Note the use of \( \text{figure}(N) \), which opens figure window number \( N \) for plotting.

### 3.8 Matrix operations

Matlab will, as mentioned in section 3.7, calculate with normal matrix operations (linear algebra) unless we use a dot before the operators. Since I rarely use linear algebra myself, I will only briefly show an example of how to perform such calculations. Assume that we are given the following equations:

\[
x_1 + 3x_2 + 7x_3 + 2x_4 + 19x_5 = 12
\]
Figure 4: The functions $XY$ and $\cos(XY)$. The commands used to generate the matrices are shown in section 3.7.

\[
\begin{align*}
2x_1 + 2x_2 + 1x_3 + 9x_4 + 12x_5 &= 10 \\
5x_1 + 3x_2 - 7x_3 + 2x_4 - 23x_5 &= 0 \\
9x_1 + 5x_2 + 9x_3 + 4x_4 + 11x_5 &= 2 \\
2x_1 + 1x_2 - 2x_3 + 0.5x_4 + 2x_5 &= 6.
\end{align*}
\]

You can probably solve these equations manually in a day, but with Matlab we can create the matrices $A=[1 3 7 2 19; 2 2 1 9 12; 5 3 -7 2 -23; 9 5 4 11; 2 1 -2 0.5 2]$ and $B=[12; 10; 0; 2; 6]$. We find the determinant to $A$ with $\text{det}(A)$. Since the determinant is -20642, we know that the equations have a solution, and it can easily be found with

\[
\gg X=A^{-1}*B,
\]

which gives $x_1$ to $x_5$ expressed by the vector $X$:

\[
X = \\
\begin{bmatrix}
-1.0087 \\
3.6393 \\
-1.5448 \\
-0.2413 \\
0.7046
\end{bmatrix}
\]

### 3.9 Flow control: for, if and while

In all programming languages the ability to control flow of data is essential. In Matlab this is done with the commands for, if and while. Here are some examples:

\[
\gg for \; i=1:10,
\]

19
if i<5  a(i)=i*2;
elseif i==5  a(i)=i;
else  a(i)=i/2;
end;
end;

This may not be a very elegant program but it demonstrates the use of for and if. The matrix a becomes [2 4 6 8 5 3 3.5 4 4.5 5].
The counter i will now be 10. A while sequence can be

while i > 5,
a(i)=i*2;
i=i-1;
end;

which changes the matrix a to [2 4 6 8 5 12 14 16 18 20].

Warning! The amount of for loops used in a program is usually inversely proportional to how long you have worked with Matlab. There are often ready built-in functions in Matlab that can replace your for loops. Since for loops are not very efficient, try to keep the use of such loops to a minimum. For, as the producer assert, "life is too short for writing for loops."

3.10 Exercises to section 3

We have now covered the basic functions concerning matrices and matrix operations. Here are a few exercises exploring this further:

1. Start by creating a vector X with values from 0 to 100 at intervals of 0.1, ie X=0, 0.1, 0.2,
   ... , 100.

   (a) What is the 20th prime number in X?
   (b) How many prime numbers are less than 1 million (hint: help isprime)?
   (c) Can you factorize 649693230? (that is find a sequence of prime numbers resulting
       in this number when multiplied together.)

2. Create the vector X=[65 114 101 32 121 111 117 32 99 114 97 122 121 63 10]; This
   seemingly random sequence of numbers is really a message since each number represent a
   character or a symbol. Matlab, like most programming languages, uses the ASCII code.
   Every character in this system is assigned a value between 0 and 255. All strings in Matlab
   can therefore be looked upon as vectors with numbers that Matlab interprets as characters.
   If we want to convert numbers to characters, we can use the char function.

   (a) Which sentence is represented by X?
   (b) What is the ascii-code for 'Enter'?
   (c) In the same way as we use find for numbers, we can use findstr to find positions of
       text. What is the position of the 'r' in the text above?
3. In Geophysics we often need to work with 3 or 4 dimensional matrices. The matrices may be measurements such as global atmosphere data or data generated by models. Here is an example where we create a 3D space spanned by the axes X, Y and Z. The first axis has values from 0 to 360 with steps of 20, the second axis from -90 to 90 with steps of 20 and the third axis from 1990 to 1999.99 with steps of 1/12: \([X, Y, Z] = \text{meshgrid}(0:20:360, -90:20:90, 1990:1/12:1999.99)\); Then we generate data, \(data = \sin(X/1000) \cdot \cos(Y/100) \cdot \tanh(Z/2000)\);

(a) What is the size of the matrix \(data\)? What is the minimum, maximum, and mean value? In what position is the maximum value?

(b) Assume that the first two dimensions represent longitude and latitude, the third dimension is time and that the \(data\) are a scalar field, for instance global monthly mean temperatures for a decade. How will you calculate the mean for each year?

(c) We often make anomalies of the data (deviation from the climate mean) by removing the mean for every month. That is, for every January we remove the mean of all the January months in the data (in this case 10 Januaries) and likewise for the other months. What is the anomaly in the first element in the matrix, and in what position is this value (x coordinate, y coordinate and time)?

4. A general equation \(\alpha x + \beta y + \gamma z = c\), where \(x\), \(y\), and \(z\) are variables and the remaining are constants, span a surface in space. We assume that we have the 3 surfaces \(x + 3y + 7z = 12\), \(2x + 2y + z = 10\) and \(5x + 3y - 7z = 6\).

(a) Use \(\text{meshgrid}\) to span a space where \(x\) and \(y\) have values from -20 to 20 with steps of 5. Plot the 3 surfaces by using \(\text{surf}\) (Hint: You can change the color of the plot by typing \(h=\text{surf}(x,y,z)\); \(\text{set}(h, 'color', 'r')\));

(b) How can you test whether the surfaces are parallel?

(c) The three surfaces intersect at one point. Find this point!
4 To read and write data

We have now covered some Matlab basics; how to use the Command Window and how to generate and make use of simple matrices. It is therefore time to turn our attention to more practical aspects such as how to read your data in Matlab. These data may be observations from the atmosphere or ocean, or the results from numerical simulations. We will therefore explain how to read ascii-data, with and without a header, and how to read binary data.

4.1 Reading ascii data without header

We start by typing clear, ensuring that our workspace is empty. Then we will show how to read the data from two files where the data are stored as plain text (ascii) in a table without header.

4.1.1 Reading CTD data

We want, for instance, to read the data in the file ctd.dat in the directory /home2/tore/Mat/. If you open the file in a text editor, you will see that it contains 5 columns and 2649 rows (lines) and that the data are separated by a comma. These data are from the Fram Strait in September 1995. The columns represent pressure, salinity, potential density ($\sigma_\theta$), potential temperature and number of CTD measurements for each depth. The first 3 rows look like this:

\[
\begin{align*}
1.0, & \quad 34.679, \quad 27.428, \quad 4.940, \quad 50 \\
2.0, & \quad 34.813, \quad 27.529, \quad 4.981, \quad 664 \\
3.0, & \quad 34.794, \quad 27.514, \quad 4.982, \quad 39
\end{align*}
\]

To read these data in Matlab, we use the load function

\[
\texttt{load /home2/tore/Mat/ctd.dat}
\]

If you now write whos (section 2.8), you will see that we have the variable ctd in workspace, containing 5 columns and 2649 rows. Note that the name of the variable is given by the filename. Note also that the load function ignores the commas as long as the data are separated with empty space as well. In fact load ignores all text between the data points. However, if the data are separated by comma and no empty space, we will have to use the function dlmread.

\[
\texttt{data=dlmread('filename';','');}
\]

Note also that if the file has no extension, we would have to include the -ascii option when using load. Otherwise Matlab assumes that the file is of a special binary format, which Matlab efficiently can read and write, but can only be used by Matlab (.mat files).

If we want to split the ctd variable into pressure, salinity, density, temperature and number of observations, we can write:

\[
\texttt{pressure=ctd(:,1); salinity=ctd(:,2); density=ctd(:,3); temp=ctd(:,4); nobs=ctd(:,5).}
\]

In order to visualize the data, we plot salinity, temperature and density in three figures next to each other. This is easily done in Matlab by splitting the figure into 1x3 areas with the function subplot

\[
\texttt{subplot(1,3,1); plot(salinity,-pressure);}
\]
\[
\texttt{subplot(1,3,2); plot(temp,-pressure);}
\]

\[
\texttt{subplot(1,3,3); plot(nobs,-pressure);}
\]
Figure 5: CTD data from the Fram Strait. In (a) salinity, temperature and density are plotted as a function of pressure (depth). In (b) the temperature is plotted against salinity. The data are stored in the file /home2/tore/Mat/ctd.dat, and they are plotted with the commands in section 4.1.1.

```matlab
>> subplot(1,3,3); plot(density,-pressure);
```

The results are shown in figure 5a. If you rather want to plot the data in a TS-diagram, with salinity along the first axis and temperature along the second, you can do this with

```matlab
>> clf; plot(salt,temp);
```

which produces the plot shown in figure 5b. Before plotting the data we write clf (clear-figure). Without this command the new plot would be a subplot in the figure window.

### 4.1.2 Reading gridded topography data

Another example of plain text is the file iceland.dat placed in the same directory. This is the gridded topography data of Iceland, where the data are stored in a 49*145 matrix. Latitudes are from 63N to 73N and longitudes from 24W to 13W. Grid distance is 5 minutes (1/12 degree) in both directions. We first load the data with

```matlab
>> load /home2/tore/Mat/iceland.dat
```

and make sure with the whos function that we now have a 49*145 matrix called Iceland. To plot this it is convenient to use the meshgrid function introduced in section 3.7. We therefore write

```matlab
>> [X,Y]=meshgrid(-25:1/12:-13,63:1/12:67);
>> contourf(X,Y,Iceland,[0:100:2000]);
```

The function contourf is similar to surf, and will be shown in section 5. It plots contours at intervals determined by the vector in the argument, in this case for every 100m. The plot is
shown in figure 6a. If you are more interested in the ocean depths around Iceland, just change the vector of the argument:

```matlab
>> contourf(X,Y,iceland,[-2000:100:0]);
```

The result is shown in figure 6b. The color scale is adjusted with the command `colormap(hot)`. Read more about the use of color in section 5.2.

### 4.2 Reading ascii-data with header

The data are often stored in files containing both text and data values. There is typically a header with information about the data. One such example is the file `/home2/tore/ctd1.dat`, which is identical to the one in Section 4.1, except for the header. Except for 10 empty lines at the top, the beginning of the file looks like this:

```
Vessel: lance        | Cruise #: 7          |
Sta.#: 105          | Operator: Tore      |
Inst. #: IM961150   | Direct.: D          |
CAL Extn: C00       | Date: 04-SEP-1995   |
St.Lat.: 78 :0:12.00 | En.Lat.: 78:00:06   |
St.Long: 4 :57:2.00  | En.Long: 5:00:10    |
Depth: 2630 Meters   | Obs.: 95519         |
AverOver: 1 DBars    | ScanRate: 31.25 Hz   |
PRES Min: 1          | PRES Max: 2649      |
SALT Min: 34.67885   | SALT Max: 35.12251  |
```

Figure 6: (a) The topography of Iceland. (b) Ocean depths around Island. Data are from `/home2/tore/ctd1.dat` and are plotted with the commands in section 4.1.2.
Furevik: An introduction to Matlab

SGTH* Min: 27.42771  |  SGTH* Max: 28.17939  |
PTEMP* Min: -1.09712  |  PTEMP* Max: 5.760592  |

<table>
<thead>
<tr>
<th>PRES*</th>
<th>SALT*</th>
<th>SGTH*</th>
<th>PTEMP*</th>
<th>NOBS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0,</td>
<td>34.679,</td>
<td>27.428,</td>
<td>4.940,</td>
<td>50</td>
</tr>
<tr>
<td>2.0,</td>
<td>34.813,</td>
<td>27.529,</td>
<td>4.981,</td>
<td>664</td>
</tr>
<tr>
<td>3.0,</td>
<td>34.794,</td>
<td>27.514,</td>
<td>4.982,</td>
<td>39</td>
</tr>
</tbody>
</table>

This file is impossible to read with both load and dlmread. We need a more sophisticated tool.

4.2.1 Reading file with uninteresting header

If we just want to extract the data, it can be done with the function textread. This function can be used with several options which you can look at yourself by using the help function. To read the file ctd1.dat, type:

```matlab
>> [pressure,salinity,density,temp,nobs]=textread('/home2/tore/Mat/ctd1.dat',...  
   '%f %f %f %f %f %f %f %f %f %f %f',headerlines',25);
```

This may look somewhat cryptic, but you have 'told' the function textread to read data from ctd1.dat, and the data to be read are columns with real floats, strings (comma), floats, strings, floats, strings, float type number, strings and integers. Note that the sign '%f' ensures that the data are not returned by the function. That is, the function above returns only 5 columns: pressure, salinity, density, temperature and number of observations. The last part of the command, 'headerlines',25, tells Matlab to ignore the first 25 lines. Note that long Matlab commands may be written over several lines by using ... at the end of the line.

4.2.2 Reading file with interesting header

If we want to read many files where the number of 'headerlines' vary, or we want to extract information from this text, the approach above is not adequate. I unfortunately don't know any elegant way of doing this, but the approach below does at least work

```matlab
>> fid=fopen('/home2/tore/Mat/ctd1.dat','r');
>> dummy=fread(fid,'char');
>> fclose(fid);
```

On the first line we open the file ctd1.dat for reading ('r' is short for read), and a file identification number is given to fid (the same as unit number in Fortran). On the second line we read all the data as ascii-code, that is, as numbers between 0 and 127, and finally in the third line we close the file again.

So far so good. We have now retrieved the data and stored them as numbers. To display them as text, we use the function char that converts ascii-code to characters. Try typing

```matlab
>> char(dummy(1:1400))
```

This lists the first 1400 characters. This is more or less the same data as shown above. Note the use of the apostrophe '. It is used to transpose matrices. With whos you can see that dummy
is a 112523x1 matrix (column vector), so to print the data in the same way as they will appear in a texteditor, the matrix must be transposed to a 1x112523 matrix. Assume that we want to extract the following information from the header: date, time, and position (longitude and latitude). We first convert \texttt{dummy} to a row of characters,

\begin{verbatim}
\texttt{c=char(dummy');}
\end{verbatim}

and then use \texttt{findstr} to find the positions of the variables we want:

\begin{verbatim}
\texttt{date_index=findstr('Date:',c);
  time_index=findstr('St. Time',c);
  lon_index=findstr('St.Lon',c);
  lat_index=findstr('St.Lat',c);
}\end{verbatim}

Now we have found the positions in the character vector \texttt{c} to the first character in all the strings we searched for. ie the position to 'D' in 'Date' (679) is stored in the variable \texttt{date_index}. We can then find the date with

\begin{verbatim}
\texttt{date=c(date_index+6:date_index+16)};
\end{verbatim}

We need to know the number of characters the date uses in the file. The same procedure is repeated for the other variables

\begin{verbatim}
\texttt{time=c(time_index+9:time_index+16);
  lon=c(lon_index+8:lon_index+20);
  lat=c(lat_index+8:lat_index+20)};
\end{verbatim}

The next step is to extract the observed data from the text string \texttt{c}. We first find the position of 'NOBS'

\begin{verbatim}
\texttt{nobs_index=findstr('NOBS',c)};
\end{verbatim}

'NOBS' is followed by the data. They are now stored as a long string, but can easily be converted to a matrix of numbers with the function \texttt{str2num}. To retrieve the data, you can type

\begin{verbatim}
\texttt{ctd=str2num(c(nobs_index+5:end))};
\end{verbatim}

Note the use of \texttt{end}. We have thus finally retrieved the data we want, using a total of 14 command lines. 5 of them could have been included among the others, for instance by finding the date with the command

\begin{verbatim}
\texttt{date=c(findstr('Date:',c)+6:findstr('Date:',c)+16)};
\end{verbatim}

This may save you some writing. However, such compressed functions are difficult to follow.

### 4.3 Reading binary files

The files read in the two sections above were all of ascii type. That is, they can be displayed as plain text in a texteditor. Now we want to look at how to read binary data.

#### 4.3.1 Reading simple binary data

In order to demonstrate how to read binary data, I have made a binary version of the topography data of Iceland, called \texttt{iceland.bin} Note that this file requires only 1/8 of the storage compared
to the ascii version, and therefore explains why the binary format is very convenient to use. The
time to read and write the data is correspondingly reduced.

We retrieve the data with the `fread` function, also used in section 4.2:
```matlab
>>> fid=fopen('/home2/tore/Mat/iceland.bin', 'r');
>>> data=fread(fid, 'short');
>>> fclose(fid);
```
The option 'short' in `fread` ensures that the data are read as short integers. That is, every
number uses 2 bytes (16 bits) of storage. If you issue the wrong format, no error message is
displayed, but the retrieved data will make no sense. The `whos` function shows that the data
in `data` are now stored as a long column vector 7105x1. To get them into the same form as in
section 4.1, we need to alter the shape of the matrix. We want the first 49 data points in the
first column, the next 49 in the second column, and so on. Thankfully there is, even for this,
such a function in Matlab:
```matlab
>>> iceland2=reshape(data, 49, 145);
```
You may try to read the `iceland.dat` file once more, and then find out whether the two matrices
`iceland` and `iceland2` are identical:
```matlab
>>> isequal(iceland, iceland2)
```
This function should return the answer 1, confirming that they are identical.

### 4.3.2 Reading unformatted binary data saved in Fortran

Note that for unformatted binary files written by Fortran (and probably many other programs)
there is an integer at the first and last line of the file. This is the number of bytes of data
stored in the file. If the data are printed to the file with more than one print command (that is,
with the use of more than one write sentence), there is such an integer for each time the print
command is issued. Here is an example of print commands in a Fortran 90 routine:

```fortran
CHARACTER(8) :: cfield
REAL*8, DIMENSION(130,120) :: longitude, latitude
OPEN(3, FILE='grids', FORM='UNFORMATTED')
WRITE(3, cfield)
WRITE(3, longitude)
WRITE(3, latitude)
```

What Fortran prints to the file `grids` is a string of 8 characters and two tables with 130x120
64-bits floats. In order to read these data in Matlab, we need to do the following
Figure 7: The data in /home2/tore/ Mat/grids are retrieved and plotted with the commands in section 4.3.2. This grid is used in test runs of a global coupled atmospheric and ocean model. The plot on the front cover is a result from these test runs.

```matlab
>> fid=fopen('grids', 'r'); % open file to read
>> fread(fid, 1, 'int32'); % returns 8, number of bytes in cfield
>> c=fread(fid, 8, 'char'); % returns cfield (8 bytes string)
>> fread(fid, 1, 'int32'); % returns 8, number of bytes in cfield
>> fread(fid, 1, 'int32'); % returns 124800, number of bytes in longitude
>> I=fread(fid, 130*120, 'float64'); % returns longitude (15600 bytes floats)
>> fread(fid, 1, 'int32'); % returns 124800, number of bytes in longitude
>> fread(fid, 1, 'int32'); % returns 124800, number of bytes in latitude
>> J=fread(fid, 130*120, 'float64'); % returns latitude (15600 floats)
>> fread(fid, 1, 'int32'); % returns 124800, number of bytes in latitude
>> fclose(fid); % closes the file
```

The file grids is in the directory /home2/tore/ Mat/, so you may try to read this file with the commands above (remember to give the whole filename in the fopen function). The function char(c') will reveal that the string reads oceangrd. By plotting the data with plot(I, J, '.'), we get the plot shown in figure 7.
4.4 Saving ascii-data

Having understood how to read ascii and binary data, it should be easy to save the data in these two formats. The opposite function of load is save. To demonstrate the use of this function, we retrieve the CTD data once more

```
>> load /home2/tore/Mat/ctd.dat;
```

Then we want to save them to the file ctd.asc in the same directory. This is done with

```
>> save /home2/tore/Mat/ctd.asc ctd -ascii;
```

Note the use of the -ascii option. If not used, Matlab will save the data in its own binary format. This is ok of course if you are only going to use the data in Matlab. However, if you want to retrieve the data in Fortran or C++, you are in trouble. By comparing the new file ctd.asc with the old ctd.dat, you will discover that ctd.asc is almost twice as large as ctd.dat. This is because Matlab by default saves the data in the following format

```
1.0000000e+00  3.4679000e+01  2.7428000e+01  4.9400000e+00  5.0000000e+01
2.0000000e+00  3.4813000e+01  2.7529000e+01  4.9810000e+00  6.6400000e+02
3.0000000e+00  3.4794000e+01  2.7514000e+01  4.9820000e+00  3.9000000e+01
```

If you want to separate the data with commas, you can do this with the function dlmwrite:

```
>> dlmwrite('/home2/tore/Mat/ctd.asc','ctd','
```

The first three lines are now

```
1,34.679,27.428,4.94,50
2,34.813,27.529,4.981,664
3,34.794,27.514,4.982,39
```

This is a more compressed format compared to the save function above. Warning: dlmread is SLOW.

The format above might not always be very useful. If you want them in a more readable form, you can do this with the fprintf function. This function is more or less the opposite of the textread function used in section 4.1 (see also the similar function fscanf). The data can be saved in the following way

```
>> fid=fopen('/home2/tore/Mat/ctd.asc','w');
>> fprintf(fid,'%4d %6.3f %6.3f %6.3f %3d\n',ctd);
>> fclose(fid);
```

This function is lightning compared to the dlmwrite above, and the data are stored in the format we want:

```
1 34.679 27.428 4.940 50
2 34.813 27.529 4.981 664
3 34.794 27.514 4.982 39
```

We told the function to save to the file ctd.asc, and that the data should be stored as: integers (4 spaces), floats (6 spaces, 3 behind comma), floats, floats, integers (3 spaces) and lineshift. This sequence is repeated until all the data are printed to the file. Note that we had to make use of the transpose operator'.
4.5 Saving binary data

Data in binary format may be written in the same way as we read them. The only difference being that we need to open the files for writing, and use `fwrite` instead of `fread`. We can first try to save the `ctd` matrix in binary format:

```
% fid=fopen('/home2/tore/Mat/ctd.bin','w');
% fwrite(fid,ctd(:,1), 'int16');
% fwrite(fid,ctd(:,2:4), 'float32');
% fwrite(fid,ctd(:,5), 'int16');
% fclose(fid);
```

The first column (pressure) is stored as 16 bits integers (2 bytes pr number). Then the following three columns (salinity, density and temperature) are stored as 32 bits floats, and finally the last column (number of observations) as 16 bits integers. Note that a total of 16 character spaces are used for each line in the `ctd` matrix, as opposed to 32 in the ascii-version with the function `fprintf`. The file size is therefore reduced to half, 42384 character spaces compared to 84768 above. In other words, the binary format is very useful for large amounts of data, both in terms of storage and fast access to the file. But the cost is that we are unable to display them in texteditors, and in order to read them in other programs, we need to know how they are saved.

4.6 Exercises to section 4

We will now practice how to read and write data in Matlab. The data to be used are stored in the file `/home2/tore/Mat/kule.dat`. The first five lines are

```
X =
```

```
   0  0  0  0  0  0  0  0
-0.5000 -0.2500  0.2500  0.5000  0.2500 -0.2500 -0.5000
-0.8660 -0.4330  0.4330  0.8660  0.4330 -0.4330 -0.8660
```

Then there are four more lines with numbers, then two empty lines and then the same for `Y` and `Z`.

1. Load the file into a texteditor and remove all text and empty lines, leaving you with just 21 rows with 7 numbers in each row. Save this file as `kule1.dat`. Load the data into the Matlab workspace. Let the matrix `X` consist of the first 7 rows, `Y` the following 7 and `Z` the last 7. Plot the data using the function `surf`.

2. Read the 7 columns in `X` from `kule.dat` by using the function `textread`.

3. Let us assume that we have a 1000 similar files to be read where the number of text and data lines vary from file to file. It is clearly then too cumbersome to edit every single file. Use therefore the procedure shown below.

   (a) Read the whole file into a character string `c` by using the `fread` function.
Furevik: An introduction to Matlab

(b) The text in c is now stored as a sequence of numbers. How will you proceed to convert c to readable text?

(c) Use the functions findstr and str2num to extract the data from c and store them in matrices X, Y and Z.

4. Then we will look at different ways of saving the data. Save the data in the variable X to the file test.dat where

(a) the first two lines in test.dat look like

```
0.0000000e+00  0.0000000e+00  0.0000000e+00  0.0000000e+00...
-5.0000000e-01  -2.5000000e-01  2.5000000e-01  5.0000000e-01...
```

What is the size of this file? How many bytes pr.

(b) the first two lines in test.dat look like

```
; ; ; ;
-0.5; -0.25; 0.25; 0.5; 0.25; -0.25; -0.5
```

What is the size of this file?

(c) the first two lines in test.dat look like

```
0.000  0.000  0.000  0.000  0.000  0.000
-0.500  -0.250  0.250  0.500  0.250  -0.250  -0.500
```

What is the size of this file?

(d) test.dat shall be of binary format and all numbers saved as floats with double precision (64 bits)? What is the size of this file?

(e) test.dat shall be of binary format and all numbers saved as floats with single precision (32 bits)? What is the size of this file?

(f) test.dat shall be of Matlabs own binary format (.mat)? What is the size of this file? Why do you think it requires larger storage than the double precision above?
5 Graphics

A major reason for using Matlab and arguably its main strength is the ability to quickly visualize data with nice graphics. In the following subsections we will discuss some routines for plotting data in one, two and three dimensions.

5.1 Object-oriented graphics

The graphics in Matlab is object-oriented and organized in a hierarchy. At the highest level is the figure window. All other graphics objects are descendents of (and display in) figure windows. The axes are 'children' of figures and are 'parent' of lines, surfaces and text. Let us show an example where we plot salinity as a function of depth (pressure) from the CTD data

\[
\text{load } /\text{home}/\text{tore}/\text{Mat}/\text{ctd.dat};
\]

\[
\text{plot(ctd(:,2),-ctd(:,1));}
\]

The plot is shown in figure 8. We have now generated objects at 3 different levels:

5.1.1 The figure level

The figure window is at the top of the hierarchy, and the first figure window will always have object number 1. The appearance of this window and its placement on the screen and printout, are all parameters attached to this object. All together, there are 62 parameters controlling the figure window. There are two commands used to change the default settings. Firstly, the function \texttt{get} which needs an object number and a parameter name as input. Secondly, the function \texttt{set} which you will have to give an object number, a parameter name and a parameter value. We will here take a closer look at both of these commands.

We use the function \texttt{get} to display the current values of the parameters. If you type:

\[
\gg \text{get(1)}
\]
Table 1: Parameters attached to the figure window.

or

\[ \texttt{get(gcf)} \]

you will get the name of all the 62 parameters and their current values. (table 1). Note that the two expressions above are analogs since \texttt{gcf} returns the object number of the current figure. (get-current-figure). It will take too long to go through all of the 62 parameters, but we will discuss some of them.

The third parameter in the list is

\texttt{Color = [0.8 0.8 0.8]}

telling us that the color of the figure window is greyish (more about colors in section 5.2). We can also obtain this information by using the \texttt{get} function

\[ \texttt{get(gcf,'Color')} \]

If we want another color, we can change the parameter value by using the \texttt{set} function. Try with

\[ \texttt{set(gcf,'Color',[0.2 0.2 0.2])} \]

and the color of the figure window turns almost black. Another parameter is \texttt{Name} with an empty string as default. If you want a name on your figure window (practical when there are lots of figures on the screen at the same time), you can write

\[ \texttt{set(gcf,'Name','CTD station nr 1')}; \]

and the text is displayed at the top of the figure window. If you want to move the figure on the screen or alter the size of it, you can do this manually with the mouse or by giving new
parameter values

\[
\text{set(gcf,'Position',[100 150 600 400])};
\]

where the first two numbers represent the position of the bottom left corner of the figure, and the last two numbers are width and height of the window.

All the parameters of the figure window can be adjusted in the same way. Finally, we will look at the parameter called `Children`. This parameter gives an object number to all the 'children' of the figure. In this case there is only one, namely the axes. If you want the object number of the axes and store it in the variable `axesnr`, write

\[
\text{axesnr}=\text{get(gcf,'Children')};
\]

### 5.1.2 The axes level

The properties of the axes are, just like the figure window, determined by many parameters, in this case 90. The functions `get` and `set` are used in exactly the same way as for the figure window, but instead of the object number of the figure window we will now, of course, have to use the object number of the axes (get-current-axes). To list all the parameters, write

\[
\text{get(axesnr)}
\]

or

\[
\text{get(gca)}
\]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AmbientLightColor</td>
<td>[1 1 1]</td>
</tr>
<tr>
<td>Box</td>
<td>on</td>
</tr>
<tr>
<td>CameraPosition</td>
<td>[34.9 -1500 17.3205]</td>
</tr>
<tr>
<td>CameraPositionMode</td>
<td>auto</td>
</tr>
<tr>
<td>CameraTarget</td>
<td>[34.9 -1500 0]</td>
</tr>
<tr>
<td>CameraTargetMode</td>
<td>auto</td>
</tr>
<tr>
<td>CameraUpVector</td>
<td>[0 1 0]</td>
</tr>
<tr>
<td>CameraUpVectorMode</td>
<td>auto</td>
</tr>
<tr>
<td>CameraViewAngle</td>
<td>[6.60861]</td>
</tr>
<tr>
<td>CameraViewAngleMode</td>
<td>auto</td>
</tr>
<tr>
<td>CLim</td>
<td>[0 1]</td>
</tr>
<tr>
<td>CLimMode</td>
<td>auto</td>
</tr>
<tr>
<td>Color</td>
<td>[1 1 1]</td>
</tr>
<tr>
<td>CurrentPoint</td>
<td>[2 by 3 double array]</td>
</tr>
<tr>
<td>ColorOrder</td>
<td>[7 by 3 double array]</td>
</tr>
<tr>
<td>DataAspectRatio</td>
<td>[1 6000 4]</td>
</tr>
<tr>
<td>DataAspectRatioMode</td>
<td>auto</td>
</tr>
<tr>
<td>DrawMode</td>
<td>normal</td>
</tr>
<tr>
<td>FontAngle</td>
<td>normal</td>
</tr>
<tr>
<td>FontName</td>
<td>Helvetica</td>
</tr>
<tr>
<td>FontSize</td>
<td>[10]</td>
</tr>
<tr>
<td>FontUnits</td>
<td>points</td>
</tr>
<tr>
<td>FontWeight</td>
<td>normal</td>
</tr>
<tr>
<td>GridLineStyle</td>
<td>-</td>
</tr>
<tr>
<td>Layer</td>
<td>bottom</td>
</tr>
<tr>
<td>LineStyleOrder</td>
<td>-</td>
</tr>
<tr>
<td>LineWidth</td>
<td>[0.5]</td>
</tr>
<tr>
<td>NextPlot</td>
<td>replace</td>
</tr>
<tr>
<td>PlotBoxAspectRatio</td>
<td>[1 1 1]</td>
</tr>
<tr>
<td>PlotBoxAspectRatioMode</td>
<td>auto</td>
</tr>
<tr>
<td>Projection</td>
<td>orthographic</td>
</tr>
<tr>
<td>Position</td>
<td>[0.13 0.11 0.775 0.815]</td>
</tr>
<tr>
<td>TickLength</td>
<td>[0.01 0.005]</td>
</tr>
<tr>
<td>TickDir</td>
<td>in</td>
</tr>
<tr>
<td>TickDirMode</td>
<td>auto</td>
</tr>
<tr>
<td>Title</td>
<td>[152]</td>
</tr>
<tr>
<td>Units</td>
<td>normalized</td>
</tr>
<tr>
<td>View</td>
<td>[0 90]</td>
</tr>
<tr>
<td>XColor</td>
<td>[0 0 0]</td>
</tr>
<tr>
<td>XDir</td>
<td>normal</td>
</tr>
<tr>
<td>XGrid</td>
<td>off</td>
</tr>
<tr>
<td>XLabel</td>
<td>[153]</td>
</tr>
<tr>
<td>XAxisLocation</td>
<td>bottom</td>
</tr>
<tr>
<td>XLim</td>
<td>[34.65 35.15]</td>
</tr>
<tr>
<td>XLimMode</td>
<td>auto</td>
</tr>
<tr>
<td>XScale</td>
<td>linear</td>
</tr>
<tr>
<td>XTick</td>
<td>[1 by 11 double array]</td>
</tr>
<tr>
<td>XTickLabel</td>
<td>[11 by 5 char array]</td>
</tr>
<tr>
<td>XTickLabelMode</td>
<td>auto</td>
</tr>
<tr>
<td>XTickLabel</td>
<td>[152]</td>
</tr>
<tr>
<td>XTickLabelMode</td>
<td>auto</td>
</tr>
<tr>
<td>YColor</td>
<td>[0 0 0]</td>
</tr>
<tr>
<td>YDir</td>
<td>normal</td>
</tr>
<tr>
<td>YGrid</td>
<td>off</td>
</tr>
<tr>
<td>YLabel</td>
<td>[154]</td>
</tr>
<tr>
<td>YAxisLocation</td>
<td>left</td>
</tr>
<tr>
<td>YLim</td>
<td>[−3000 0]</td>
</tr>
<tr>
<td>YLimMode</td>
<td>auto</td>
</tr>
<tr>
<td>YScale</td>
<td>linear</td>
</tr>
<tr>
<td>YTick</td>
<td>[1 by 7 double array]</td>
</tr>
<tr>
<td>YTickLabel</td>
<td>[152]</td>
</tr>
<tr>
<td>YTickLabelMode</td>
<td>auto</td>
</tr>
<tr>
<td>YTickLabel</td>
<td>[152]</td>
</tr>
<tr>
<td>YTickLabelMode</td>
<td>auto</td>
</tr>
<tr>
<td>YTickLabel</td>
<td>[152]</td>
</tr>
<tr>
<td>YTickLabelMode</td>
<td>auto</td>
</tr>
<tr>
<td>ButtonDownFcn</td>
<td>ButtonDownFcn</td>
</tr>
<tr>
<td>Children</td>
<td>[1.00098]</td>
</tr>
<tr>
<td>Clipping</td>
<td>on</td>
</tr>
<tr>
<td>CreateFcn</td>
<td>CreateFcn</td>
</tr>
<tr>
<td>DeleteFcn</td>
<td>DeleteFcn</td>
</tr>
<tr>
<td>BusyAction</td>
<td>queue</td>
</tr>
<tr>
<td>HandleVisibility</td>
<td>on</td>
</tr>
<tr>
<td>HitTest</td>
<td>on</td>
</tr>
<tr>
<td>Interruptible</td>
<td>on</td>
</tr>
<tr>
<td>Parent</td>
<td>[1]</td>
</tr>
<tr>
<td>Selected</td>
<td>off</td>
</tr>
<tr>
<td>SelectionHighlight</td>
<td>on</td>
</tr>
<tr>
<td>Tag</td>
<td>axes</td>
</tr>
<tr>
<td>Type</td>
<td>axes</td>
</tr>
<tr>
<td>UIContextMenu</td>
<td>[]</td>
</tr>
<tr>
<td>UserData</td>
<td>[]</td>
</tr>
<tr>
<td>Visible</td>
<td>on</td>
</tr>
</tbody>
</table>

**Table 2: Parameters attached to the axes.**
The list of parameters is shown in table 2. The color of the axes window is given by Color. If you want red instead of black write

```matlab
set(gca,'Color',[1 0 0]);
```

Here are some other parameters I often adjust:

```matlab
set(gca,'Fontangle','italic');  % Ital. text fonts
set(gca,'Fontsize',15);         % Larger text fonts
set(gca,'LineWidth',2);         % Thicker lines
set(gca,'Position',[-0.5 -0.5 1 1]); % Position relative to fig. window
set(gca,'XColor',[0 0 1]);      % The color of the first axes
set(gca,'XLim',[34.5 35.25]);   % The limits of the first axes
set(gca,'XTick',[34.5:25:35.25]); % Tick position on the first axes
set(gca,'XTickLabel',...  
  [34.5:25:35.25])            % Grid parallel to the y-axes
```

Note that it is possible (and practical) to do many operations at the same time, so that the list above can be displayed with the command

```matlab
set(gca,'Fontangle','italic', 'Fontsize',15, 'LineWidth',2,...  
  'Position',[-0.5 -0.5 1 1], 'XColor',[0 0 1], 'XLim',[34.5 35.25],...  
  'XTick',[34.5:25:35.25], 'XTickLabel',...  
  [34.5:25:35.25])
```

There is also for this object a parameter called Children. In this case it is the object number of the graph itself. This number can be stored in the variable graphnr by writing

```matlab
graphnr=get(gca,'Children');
```

Note that we also have a 'parent' with object number 1. In this case it is the figure window. The axes also have 'children' which we can get hold of by using title, xlabel, ylabel and zlabel. We will return to these later.

### 5.1.3 The graph level

The lowest level of the Matlab hierarchy is the graph level. Here we can find all the objects belonging to the axes. The object number of the graph is now stored in the variable graphnr. To view the list of parameters we write as before

```matlab
get(graphnr);
```

and we get a list of 28 parameters (table 3). If you now, for instance, want the color of the graph to be black with thickness 2 mm, you can write

```matlab
set(graphnr,'Color',[0 0 0],'LineWidth',2);
```

A tip is to store the object number of the graph in a variable when you plot the graph. You can also at the same time change some of the predefined parameters in the plot function. (as done in section 3.7). Here are a couple of examples:

```matlab
h=plot(0:4,-0.5:1,'Color',[1 0 0],'LineWidth',2,'LineStyle',':');
```

Now we plotted a red graph ('r' option) with linewidth 2mm and a dashed line. At the same
time we stored the object number in the variable \( h \), often called a handle. Type \( \text{get}(h) \) to look at the handle. Note that the main colors and line type can be given as a string in the plot function. So we could have done this by typing
\[
> h = \text{plot}([c1d(1,:);],[c1d(:,1),',r-',',Linewidth',2]);
\]
The plot is shown in figure 9a.

5.1.4 Adding text

We usually want to add text to our axes. This may be a title, labels or simply some comments underneath the graph. All these objects will be children of the axes. The object numbers of the title and axis labels can be found with \( \text{get(gca)} \) as shown above. However, it is easier to define them directly with the functions \textit{title}, \textit{xlabel} and \textit{ylabel}:
\[
> ht = \text{title('CTD station in The Fram Strait')} ;
> hx = \text{xlabel('Temperature (C)')} ;
> hy = \text{ylabel('Depth (m)')} ;
\]
The results are shown in figure 9b. Note that, as we added a title and labels, the object numbers were stored in variables \( h \). They can be used to adjust the text later. If we write
\[
> \text{get(ht)} ;
\]
we will see that there are 33 different parameters we can change (table 4) Remember that for all text strings, the position is relative to the axes. So if we want a larger title and place it at the right edge of the plot, we can write
\[
> \text{set(ht,'FontSize',20,'Position',[6 125 0],'HorizontalAlignment','right')} ;
\]
With 'HorizontalAlignment' we tell Matlab that the right side of the text should be in the given position. Otherwise Matlab will centre the text around this position. Note that the position vector also has a z-value, but in this case it can be anything. The plot is shown in figure 9c.

Now we want to add some text to the plot. This can be done by using the \textit{text} function. If you know the position you want the text, you can write
Figure 9: Demonstration of the use of object numbers (section 5.1.3 and 5.1.4).

```matlab
>> h=text(6,-3000,'made by Tore Furevik');
>> set(h,'Fontsize',8,'HorizontalAlignment','right','VerticalAlignment','bottom');
```

A useful function in this context is the `ginput` function. It displays the position when you click the mouse on the plot. ie

```matlab
>> [x,y]=ginput; h=text(x,y,'Deep water');
```

Note that you should only press the mouse button once, otherwise the text is displayed several times. Alternatively, you can place the text directly with the mouse by writing

```matlab
>> h=gtext('Atlantic Water');
```

and then press the button at the required position.
And finally, the two important functions grid and hold. The grid function, as the name suggests, adds a grid to the figure. This is shown in figure 9d (perhaps not an outstanding figure, but it should illustrate the point). The hold function is used when we want to draw more than one graph on the same axes, or else the new graph will replace the old. Both of these functions are on/off switches, that is, if you type grid twice you will end up with no grid. To avoid accidents it may be wise to use the add on or off. The grid on always ensures that the grid will be displayed.

5.2 The colors in Matlab

The colors in Matlab are based on the three main colors: red, green and blue (rgb system). All colors can therefore be represented by a vector with three numbers between 0 and 1, where the first number is the amount of red, the second the amount of green and the third the amount of blue. Alternatively, you can use the pure colors (yellow, magenta, cyan, red, green, blue, white or black) as strings containing a single character (y, m, c, r, g, b, w or k).

Matlab also often uses colormaps. Colormaps are matrices with 64 colors. A summary of the different colormaps is listed by writing help graph3d. The colormaps are

Colormaps.

- hsv - Hue-saturation-value colormap.
- hot - Black-red-yellow-white colormap.
- gray - Linear gray-scale colormap.
- bone - Gray-scale with tinge of blue colormap.
- copper - Linear copper-tone colormap.
- pink - Pastel shades of pink colormap.
- white - All white colormap.
- flag - Alternating red, white, blue, and black colormap.
- lines - Colormap with the line colors.
Furevik: An introduction to Matlab

colorcube - Enhanced color-cube colormap.
vga - Windows colormap for 16 colors.
jet - Variant of HSV.
prism - Prism colormap.
cool - Shades of cyan and magenta colormap.
autumn - Shades of red and yellow colormap.
spring - Shades of magenta and yellow colormap.
winter - Shades of blue and green colormap.
summer - Shades of green and yellow colormap.

We can demonstrate the use of colormaps with a small script

```
>> [x,y]=meshgrid(0:1,0:.01:1);
>> surf(x,y);
```

where the function \( y(x,y) \) is plotted over the domain 0 to 1. Matlab automatically adds colors drawn from the colormap jet, running from blue via green to yellow and red. This is a suitable scale for plotting temperatures since it goes from a cold to a warm color. If you want to use other colormaps, use the `colormap` function. Try, for instance, to write

```
>> colormap(hot);
```

which gives colors from black via red to yellow and white. Or

```
>> colormap(cool);
```

which only uses the cold colors in blue and lilac. The first color is \([0 1 1] \) (type cool), a mix between green and blue, and the last color is \([1 0 1] \), a mix between red and blue. We can define our own colormap by making a Mx3 matrix containing numbers between 0 and 1. ie

```
>> a=rand(100,3); colormap(a);
```

which generates and uses a colormap with 100 randomly selected colors. To return to the original colormap, simply type `colormap('default'). Note that `colormap(jet)` would yield almost the same colors but the color map would now have 100 instead of 64 colors. Plots with the four different colormaps `jet` (default), `hot`, `cool` and `rand(100,3)` are shown in figure 10.

### 5.3 Two-dimensional plots

We will now discuss some different forms of two-dimensional plots. As an example, we load current speed measurements from position 79°N, 0°E in the Fram Strait. Measurements were taken from a depth of 60 m at intervals of one hour from 14/5/1985 to 11/8/1986. You should remove the variables in the workspace with the `clear` function before we start.

#### 5.3.1 Reading the data to be plotted

The first four lines of the current speed measurements look like this

```
M=3163 N= FAUVTS Z= 60 T=85 714-1930 Z= 107 PS=N7901 E00054
YMMDD-hh:mm P A U V T S
85 714-19:30 2.70 91.2 2.70 -.06 2.597 34.449
85 714-20:30 3.26 111.4 3.04 -1.19 2.508 34.449
```
Figure 10: Demonstration of the colormaps jet, hot, cool and a random distribution. The plot commands are shown in section 5.2.

Since we are not interested in the first two lines, we read the data with the `textread` function:

\[
\text{\texttt{[Time,F,A,U,V,T,S]=...}}
\]

\[
\text{\texttt{textread('}/home2/tore/Mat/mooring.dat',}'\%12c \%f \%f \%f \%f \%f',}'\texttt{\textquote{\textbackslash'\'headerlines'}}\text{\textquoteright},2);}
\]

We read the first 12 characters as a text string, then 6 columns as floats representing speed, direction, eastern and western component of the speed, temperature and salinity. The function `whos` will now display the following variables:

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>9436x1</td>
<td>75488</td>
<td>double array</td>
</tr>
<tr>
<td>F</td>
<td>9436x1</td>
<td>75488</td>
<td>double array</td>
</tr>
<tr>
<td>S</td>
<td>9436x1</td>
<td>75488</td>
<td>double array</td>
</tr>
<tr>
<td>T</td>
<td>9436x1</td>
<td>75488</td>
<td>double array</td>
</tr>
</tbody>
</table>
Time  9436x12   226464 char array
U     9436x1    75488 double array
V     9436x1    75488 double array
ctd   2649x5    105960 double array

Grand total is 183093 elements using 785352 bytes

5.3.2 The time variable

Note the variable Time, a matrix with 12 columns and 9436 rows. All the information about the time of measurements are stored in this variable, but in a form which is of no use to us. But we can split the time matrix into matrices for years, months and so on. Do this with

\[ YY = \text{str2num}(\text{Time}(:,1:2)); \]
\[ MM = \text{str2num}(\text{Time}(:,3:4)); \]
\[ DD = \text{str2num}(\text{Time}(:,5:6)); \]
\[ hh = \text{str2num}(\text{Time}(:,8:9)); \]
\[ mm = \text{str2num}(\text{Time}(:,11:12)); \]

Now we will look at two very useful functions when time is one of our dimensions. The first is \textit{datenum}, which from a given time calculates the number of days since year zero. Day 1 is midnight the first of January in year zero. The other function is \textit{datestr}, which for a given day calculates the date and time of day. These simple programs put an end to lots of small programs taking into account the different number of days in every month and so on. Matlab takes care of all that. Have you, by the way, ever tried to calculate how many days old you are? I was at the time of writing

\[ \text{datenum}(2000,3,2) - \text{datenum}(1969,8,9) \]

(11163) days old. I had my 10000 jubilee the 25th of December 1996

\[ \text{datestr} (\text{datenum}(1969,8,9) + 10000) \]

which was a wednesday (\textit{datestr}(\text{datenum}(1969,8,9) + 10000,8)). Type \textit{help datestr} to see all the 18 possible formats this function can print.

After doing this we store the time variables YY, MM, DD, hh, and mm in a variable called \textit{day}. The number of days since year zero can then be found with

\[ \text{day} = \text{datenum}(1900 + YY, MM, DD, hh, mm, 0); \]

Note that we can also use seconds as well as many other formats. To make sure everything is correctly done so far, type \textit{datestr(day(1))}, and compare it with the first dateline in the file \textit{mooring.dat} as shown above.

5.3.3 The function \textit{plot}

The most common plot function in 2 dimensions is, as the name suggests, \textit{plot}. Some examples have already been shown in earlier sections. We start off by plotting the speed \( F \) as a function of the time \( \text{day} \)

\[ h = \text{plot}(\text{day}, F); \text{ grid}; \]
The variable `day` is the number of days since year zero. In order to get some more useful units on the axes we find the day number of the first day in every month in 1985 and 1986

```matlab
> dagma=datenum(1985,1,24);
```

which returns a vector containing the 24 days. Now we can use what we learned in section 5.1, and type

```matlab
> set(gca,'XTick',dagma,'XTickLabel',datestr(dagma,’3’));
> xlabel(’Time 1985-1986’); ylabel(’Current speed’);
```

and the result is the plot shown in figure 11a. It is evident from the graph that the current speed is typically around 10 cm/s with minima close to 0 cm/s and maxima close to 55 cm/s. If we use the functions `mean`, `min` and `max` on the speed `F`, we get 14.2956 cm/s, 0.7500 cm/s and 53.6700 cm/s. Then we turn our attention towards the measured salinity and temperature. We open a new figure window and plot a TS diagram

```matlab
> figure(2);
> plot(T,S,’.’); grid;
```

The new figure window is opened with the command `figure`, and the temperature plotted as a function of salinity with dots instead of a solid line. The result is shown in figure 11b. There is obviously, by looking at the graph, a relation between temperature and salinity. The warm water indicates large contribution from Atlantic water and the cold, fresh water a large contribution from polar water.

Apparent are also a few outliers, and they are clearly not physical. They include 2 measurements where the temperature is below freezing (approx -1.9°C), and one measurement where the temperature suddenly is 7°C. Also the two measurements with salinity below 31 psu look dubious. These measurements can be located by using the `find` function. The indices of the 6 outliers can be found with

```matlab
> I=find(T<-2|T>5|S<31);
```

If we type `datestr(T(I))` we discover that the temperatures 2.0360°C, -2.4780°C and 6.9650°C (`T(I(1:3))`), and the salinities 0.4880 psu, 34.0270 psu, and 31.5740 psu (`(S(I(1:3)))`) were recorded at 29/9/1985 13:30, 14:30 and 15:30. These three recordings are obviously erroneous. We therefore set the 6 data points to non-numbers and plot the data again

```matlab
> T(I)=NaN; S(I)=NaN;
> plot(T,S,’.’); grid;
```

The new plot is shown in figure 11c. There are still probably some dubious outliers, but at least they are less obvious.

A few more simple analyses may be appropriate. We start by removing the non-numbers since they in many functions will cause problems. They can be removed with

```matlab
> dag=diag(find(~isnan(T))); T=T(find(~isnan(T))); S=S(find(~isnan(S)));
```

Note that we also remove the corresponding times in the day variable, since the variables otherwise would be of unequal length.

In order to perform a regression analysis, we first find the regression coefficients with

```matlab
> p=polyfit(T,S,1);
```
Figure 11: Demonstration of the plot function. The data are from the file /home2/tore/Mat/mooring.dat, and the commands used to make this plot are shown in section 5.3.3.

and plot the regression line with

\[ \text{hold on; h = plot(T, p(1) * T + p(2), 'r', 'Linewidth', 2);} \]

The gradient of the regression line is \( p(1) \) (0.1971) and the zero point is \( p(2) \) (33.8168). Now we can find the standard deviation and the maximum distance to the regression line:

\[ a = \text{std}(S - (p(1) * T + p(2))); b = \text{max}(\text{abs}(S - (p(1) * T + p(2)))) \]

and the results are 0.0894 and 0.6512. In other words the greatest deviation is 7.3 standard deviations from the regression line, a result larger than we should expect. As an illustration we
mark off all points further than 3 standard deviations away from the regression line
\[
\geq I=\text{find}(\text{abs}(S-(p(1)*T+p(2)))/a>3);
\]
\[
\text{hold on; plot}(T(I),S(I),'r*'); \text{hold off};
\]
The resulting 67 points are in figure 11d represented with stars.

5.3.4 Logarithmic plots

With the function plot the axes are linear by default. This can be changed by using the set command, i.e.
\[
\geq \text{set}(\text{gca},'YScale','log');
\]
But there are also functions similar to plot that do this directly. Here is an example where we calculate the energy spectrum to the speed component \(U\) from the measurements in the Fram Strait.
\[
\geq [\text{Upsd}, \text{Freq}]=\text{psd}(U,1024,24,1024,0,'\text{linear'});
\]
To those interested; the function \text{psd} (power spectra density) returns the energy per frequency unit (energy density) in the variable \text{Upsd}, and the frequency in the vector \text{Freq}. The input is the current speed component which is divided into block lengths of 1024 measurements (42.7 days). For every block the linear trend is removed and then smoothed with a Hanning window (Matlab standard) of the same length. The number 24 tells Matlab that there are 24 measurements per day, giving the right frequency in the \text{Freq} vector.

The energy density is often plotted using logarithmic axes. In Matlab this is simply done with
\[
\geq \text{loglog}(\text{Freq}, \text{Upsd}; \text{axes}([.01 12 .1 1E4]); \text{grid};
\]
\[
\text{xlabel('Frequency (oscillations pr day')}; \text{ylabel('Energy (cm*cm/frequency')};
\]
The result is shown in figure 12a. The peak in frequency of 2 oscillations per day is of course the semi-diurnal tide.

If you only want one of the axes to be logarithmic, you can use the functions \text{semilogx} or \text{semilogy}. Here is another example showing the energy density, but this time the second axis is linear:
\[
\geq \text{semilogx}(\text{Freq}, \text{Upsd}); \text{axes}([.01 12 .1 1E4]); \text{grid};
\]
\[
\text{xlabel('Frequency (oscillations pr day')}; \text{ylabel('Energy (cm*cm/frequency')};
\]
In this case the high frequency energy is almost invisible compared with the energy associated with the oscillations with long periods. (figure 12b). Note that the only difference between the plots is the scale of the second axes. Try typing \text{set}(\text{gca},'Yscale','log') to convince yourself.

5.3.5 Histogram and bar chart

The plot function is, as already mentioned, the most frequently used for plotting in two dimensions. But there are of course many other useful plot functions. Here we will mention two of them.

The function \text{hist} plots a histogram. This is very useful when we want a quick look at the distribution of data. i.e
Figure 12: Demonstration of logarithmic plots. The data are from the file /home2/tore/ Mat/mooring.dat, and are plotted with the commands in section 5.3.4.

Figure 13: Demonstration of the histogram function hist. The data are from file /home2/tore/ Mat/mooring.dat, and are plotted with the commands shown in section 5.3.5.

```matlab
> subplot(2,1,1); hist(T,-2:.5); grid; title('T distribution')
> subplot(2,1,2); hist(S,33.2:.05:34.8); grid; title('S distribution')
```
resulting in the plot shown in figure 13. An analog to plot is bar which plots the data as pillars. Here is an example where the temperature data are plotted in the top diagram and the data from the first to the third of April 1986 are plotted below.

```matlab
subplot(2,1,1); h=bar(day,T); grid; xlabel('Date 1985-1986');
set(gca,'XTick',daynum,'XTickLabel',datestr(daynum',8'));
subplot(2,1,2); h=bar(day,T); grid;
daynum2=datenum([1986,4,1,0:2:48,0,0]);
axes([daynum2(1) daynum2(end) -2 2]);
set(gca,'XTick',daynum2,'XTickLabel',num2str([0:2:48]'));
xlabel(['Hours from ' datestr(daynum2(1),1)]);
```

The result is shown in figure 14.

### 5.3.6 Plots in polar coordinates

With Matlab it is easy to make plots in polar coordinates. Assume that we want to plot the direction (A) and speed (F) of the current in the Fram Strait. In Matlab we need the direction in radians while the direction in A is given in degrees relative to north. To find the angle in radians, type

```matlab
theta=(90-A)*pi/180;
```

Then we can plot the direction as a function of speed and angle.
Figure 15: Demonstration of the polar plot functions polar and rose (section 5.3.6). The data are from the file /home2/tore/Mat/mooring.dat.

```matlab
>> figure(1); polar(theta,F,'.'); title('Velocity in polar plot');
```

The plot is shown in figure 15a. A histogram in polar coordinates can be obtained with the rose function.

```matlab
>> rose(theta,36); title('Distribution of direction ')
```

We now have a nice overview of the current direction and it is apparent that the direction is between 150 and 160 degrees (northwest) in more than 700 of the measurements (figure 15b).

### 5.3.7 Stick-plot

In oceanography so-called stick-plots are often used when plotting speed and direction. That is, for every point along the time axis an arrow is plotted where the length and angle with the time axis represent speed and direction of the measured current. This is easy to do in Matlab with the feather function. Here is a small routine creating a stick-plot for the first ten weeks, and where we divide the data into 5 plots (2 weeks = 288 data points).

```matlab
>> day=datenum(1900+YY,MM,DD,hh,mm,0);
>> for i=1:5;
    >> I=(i-1)*288+1:i*288; merke=4.5:24:288;
```
Figure 16: Demonstration of the stick-plot function feather. The data are from the file
/home2/tore/Mat/mooring.dat, and are plotted with the commands in section 5.3.7.

```matlab
〉 subplot(5,1,i); feather(U(I),V(I));
〉 set(gca,'XTick',mark,'XTickLabel',datestr(dag(I(1))+mark/24,6));
〉 axes equal; grid;
〉 end;
```

The plot is shown in figure 16.

5.3.8 Plotting two variables on the same graph

In some contexts it might be useful to have different axes in the same figure. For instance when we plot both temperature and salinity on the same graph. For this purpose there is a function called `plotyy`. A simple example:

```matlab
〉 load /home2/tore/Mat/ctd.dat;
〉 P=ctd(:,1); S=ctd(:,2); T=ctd(:,4);
〉 plotyy(P,S,P,T); grid;
```

The pressure is along the first axis while temperature and salinity are along the second axis, with two different scales (figure 17a). If we want to alter the figure, we need to get hold of the object numbers. Here is an example where we plot depth along the second axis, and use the object number to draw the graph the way we like it

```matlab
〉 [ax,h1,h2]=plotyy(S,-P,T,-P); grid;
〉 set(ax(1), 'XLim',[34.45 35.2], 'XTick',[34.45:0.5:35.2], 'XColor','b', 'YColor','k');
```

48
Figure 17: Demonstration of the `plotyy` function. The data are from the file `~/home2/tore/Mat/ctd.dat`, and are plotted with the commands in section 5.3.8.

```
>> set(ax(2), 'XLim', [-1.5 6], 'XTick', [-1.5:5:6], 'XAxesLocation', 'top', ...
   'XColor', [0 .5 0], 'YColor', 'k');
>> set(get(ax(1), 'XLabel'), 'String', 'Salinity (psu)');
>> set(get(ax(2), 'XLabel'), 'String', 'Temperature (grad C)');
>> h=text(S(2200), -P(2200), 'S'); set(h, 'Parent', ax(1), 'Color', 'b');
>> h=text(T(2200), -P(2200), 'T'); set(h, 'Parent', ax(2), 'Color', [0 .5 0]);
```

This figure is shown in figure 17b. Note that the function `plotyy` stores the two axis numbers in `ax`, while the two graph numbers are stored in `h1` and `h2`. Since the function creates two graphs on top of each other, we need to obtain the object numbers with `get(ax(1), 'XLabel')` in order to add labels. Likewise for the second graph (where the temperature is plotted). When we add text to the graphs itself, we do it the other way round. We add the text first, and then assign an object number to the text. The same method could have been applied to the axis labels.

5.4 Three-dimensional plots

We have already shown some figures in three dimensions (figures 2, 4 and 6). In this section we will study the three-dimensional plot functions in more detail. Again we can do this by using the topography data of Iceland. We load the data and create a grid with the commands

```
>> load ~/home2/tore/Mat/iceland.dat;
>> [X,Y]=meshgrid(-25.1/12:-13.63:1/12.67);
```

Then we will discuss some different functions for visualizing the data.
5.4.1 The functions contour and contourf

The functions contour and contourf are probably the most usual 3D plot functions. They draw isolines for a variable. Type help contour to see how they are used. Here is one example where we draw thin height contours with an equidistance of 100m, and a thick solid line for every 500m. The coastline is marked with a thick contour, and the depth contours with thin lines for every 250 meters.

```
>> [c1,h1]=contour(X,Y,iceland,[100:100:2000],'-'); hold on;
>> [c2,h2]=contour(X,Y,iceland,[500:500:2000],'-');
>> [c3,h3]=contour(X,Y,iceland,[0 0],'-');
>> [c4,h4]=contour(X,Y,iceland,-3000:250:-250, '-'); hold off; grid;
```

Note that in order to display the zero-line we need to type [0 0]. If we just type a single 0, Matlab will interpret this to be the number of contours to be drawn. Our preliminary plot is shown in figure 18a. Now we will use the object numbers already stored in variables. Note that each line has its own object. To see them you will therefore need to type get(h1(1)).

Here we draw thin contours for every 100m, add numbers to the contours showing the altitude above sea level, draw a thick coastline and thin depth contours. For the labels on the depth contours we use an italic font.

```
>> set(h1,'Linewidth',2);
>> h=labeled(c2,h2); set(h,'Fontsize',8);
>> set(h3,'Linewidth',2);
>> h=labeled(c4,h4); set(h,'Fontsize',8,'FontName','italic');
```
Iceland is now shown in figure 18b.

The function `contourf` is used in exactly the same way as `contour`, except that it fills the contours with colors from the current colormap (see section 5.2). This function is already shown in figure 6. Everything done with `contour` above may also be done with `contourf`. In addition it is normal to also display a colorbar on the figure indicating height or depth. The figure 19a shows a vertical version of the colorbar in Matlab. It is created with

```matlab
≫ contourf(X,Y,iceland,[0:100:2000]); grid; colorbar;
```

The colors are automatically scaled from the maxima and minima of the contour lines. Here is an example of how we instead of the default colorbar in Matlab, can specify the location of the bar in the figure:

```matlab
≫ contourf(X,Y,iceland,[0:100:2000]); grid;
≫ cb=axes('Position',[-2 0.12 0.38 0.32]); h=colorbar; axes off;
≫ mincon=0; maxcon=1900; merke=[0:200:2000];
≫ set(h,'XTick',[]);
≫ set(h,'YTick',merke/(maxcon-mincon)*64,'YTickLabel',num2str(merke));
```

This saves some space in the figure, as shown in figure 19b. The two variables `mincon` and `maxcon` were used to scale the colorbar since it as default runs from 0 to 64 colors (number of colors in the colormap).

### 5.4.2 The other 3D plot functions

The use of the other 3D plot functions are very similar to `contourf`. Here are some examples.

---

*Furevik: An introduction to Matlab*
Figure 20: Demonstration of the plot functions mesh, surf, pcolor and imagesc. The data are from the file `/home2/tore/Mat/iceland.dat`, and are plotted with the commands in section 5.4.2.

**mesh:** This function plots the data as masks. Figure 20a shows the topography data with the use of this function

```
>> mesh(X, Y, iceland);
```

which in addition to masks also plots "curtains". In many cases this is not very useful since the figures demand large storage and are slow to print. The figure is by comparison 10 times larger than the ones created with `contourf` above. As default, the data are displayed from the direction -37.5 (degrees relative to the south) and height 30 (degrees above the horizontal plane). This can be changed with the command `view(direction, height)`. One can also do this directly from the menu available in the figure window by choosing Tools and Rotate 3D. You can then choose the angle of view with the mouse.
surf: This function is similar to mesh, but there are colors between the masks. If you want to remove the masks, use the shading function where you can choose between no interpolation (flat) or interpolation (interp) between the datapoints. Here is an example of the former which also removes the axes (figure 20b):

\[ \text{surf}(X,Y,	ext{iceland}); \text{shading flat}; \text{axes off}; \]

pcolor: This function is like surf, only that the data now are viewed from above. Here is an example of a plot (figure 20c) using pcolor

\[ \text{pcolor}(X,Y,	ext{iceland}); \text{shading flat}; \text{colorbar}; \]

image: A last function we should mentioned is image which is useful for visualizing images. It is very fast even with large amounts of data. Here is an example where we use imagesc, scaling the data so that all the 64 colors of the colormap are used instead of only values between 0 and 1.

\[ \text{imagesc}(X(:,1),Y(:,1),\text{iceland},[0 1000]); \text{set(gca,'YDir','normal');} \]

The plot is shown in figure 20d.

Note that we had to change the direction of the Y-axis since Matlab plots images with the first point at the top left hand corner (as opposed to normally at the bottom left). Note also that we needed vectors as arguments spanning the whole image instead of the usual X and Y matrices. The last part of the command means that the colors are scaled from scalelevel up to 1000 meters. Everything below this becomes blue, and everything above becomes red according to the standard colormap (jet).

It may be interesting to see the amount of storage required for the four plots in figure 20. It is 1442 KB, 487 KB, 427 KB and 21 KB, and shows how efficient image or imagesc are. Large files should therefore be plotted with one of these functions. A high-resolution SAR image for instance (12.5m resolution) has 64 million points. Do not try to plot this with the mesh function!

5.5 Printing figures

It is nice to view figures on the screen but of course we also usually want to print them either to file or on paper. In Matlab the figures can simply be stored with the function print. The most useable (but storage demanding!) is arguably to store the figures as black-white or color postscript:

\[ \text{print -deps blackwhitefigurename}; \]
or

\[ \text{print -depsc colorfigurename}; \]

I strongly recommend to use extensions that are recognizable, i.e. .eps for black-white figures, and .epsc for color figures. This option stores the figures as encapsulated postscript. These figures are stored in a text format so that you can easily change colors or text in a texteditor. When only small adjustments are necessary this is a lot faster than creating new figures. The figures
may also be read straight into LaTeX and many other text editors. All figures in this document are saved with -deps or -depsc options.

Also Microsoft products such as Word and Powerpoint read postscript, although the images are usually not displayed on the screen. To display them you can create a so-called preview at the top of the postscript file. This is a binary string containing a low-resolution image. ie

```
>> print -depsc -tiff filename;
```

Now the figure in TIFF format will be displayed on the screen while the postscript figure will be printed to the printer. This is often very useful. Type help print to see the range of print features available.

### 5.6 Exercises to section 5

We will now practice the handling of graphics in Matlab. For this purpose we use a file with topography data of the Earth.

1. We start by reading the file `/home2/tore/Mat/worldmap.mat`. It contains the variables `longitude`, `latitude` and `height`.
   
   (a) Plot the data using the `contourf` function, with isolines for every 1000m.
   
   (b) Add labels and a title. A suitable title is `Topography` and labels `Longitude` and `Latitude`. Use red color and fontsize 14.
   
   (c) Plot the coastal data in the same figure, with black color and linewidth 1.
   
   (d) Store the figure in `epsc` (*encapsulated postscript color*) format and name it `worldtop1.eps`. View the file in `ghostview` without quitting Matlab.
   
   (e) Store the figure in `jpeg` format and call it `worldtop1.jpg`. View the file in `xv` without quitting Matlab.

2. We will now create a plot in spherical coordinates. Make the three variables `x`, `y` and `z` where $x = r \cos(lat) \cos(lon)$, $y = r \cos(lat) \sin(lon)$, and $z = r \sin(lat)$ where $r$ is the Earth radius (6370 km). Remember that the input arguments in the trigonometric functions must be in radians.
   
   (a) Plot the data with the function `surf`, and use shading flat and axes equal.
   
   (b) View the sphere from different angles using the `view` function. What are the input arguments to this function if we want to look straight down at Bergen?
   
   (c) Get the size and position of the axes. Then enlarge the axes so they fill the whole screen. Then hide the axes so that only the Earth is displayed.
   
   (d) Create new axes filling the entire figure window. Make yellow dots at 1000 random positions, where the size of the dots vary randomly between 0 and 10. Then hide the axes so that only the Earth and 'stars' are shown.
   
   (e) Since we created the stars after the Earth, the stars are placed on top of the sphere. You can change this by altering the sequence of the 'children' of the figure window. Finally, store the figure in `epsc` format with name `worldtop2.eps`.

54
6 Map plotting

One of the many advantages of Matlab is that it is easy to write your own scripts and functions that are transferable between Unix and PC versions. By searching the Internet you may find thousands of more or less useful Matlab routines. One of the most useful I have discovered is the map plotting routines in \textit{M.map}, made by Rich Pawlowicz at the University of British Columbia in Vancouver, Canada. His routines, together with coastline and topography data, are now stored on a shared server at the Geophysical Institute (and the Nansencenter). Here is a short demonstration of this map-package.

6.1 Choice of projection

As a start we need to choose a map projection. The possible projections can be displayed with the command \texttt{m_proj('get')}. It returns a list of 18 different projections:

Available projections are:
\begin{itemize}
\item Albers Equal-Area Conic
\item Lambert Conformal Conic
\item Mercator
\item Miller Cylindrical
\item Equidistant Cylindrical
\item UTM
\item Transverse Mercator
\item Sinusoidal
\item Gall-Peters
\item Hammer-Aitoff
\item Mollweide
\item Oblique Mercator
\item Stereographic
\item Orthographic
\item Azimuthal Equal-area
\item Azimuthal Equidistant
\item Gnomonic
\item Satellite
\end{itemize}

If you, for instance, want to choose the \texttt{Mercator} projection, just type
\texttt{m_proj('Mercator')};

You may add more options, and the possible options can be displayed with \texttt{m_proj('get', 'Mercator')}:

\begin{verbatim}
'Mercator'
'lon<itude>', ( [min max] | center)
'lat<itude>', ( maxlat | [min max])
\end{verbatim}

The default values can be viewed with \texttt{m_proj('set')}, which returns
Current mapping parameters -
Projection: Mercator (function: mp_cyl)
longitudes: -180  180
Latitudes:  -85  85

In order to plot a smaller area we may write
\[
\text{m_proj('Mercator',} \text{lon',[-90 90]},\text{lat',[0 85])};
\]
Note that the chosen projection will be used until we use the \text{m_proj} function again.

6.2 Plotting the map

Having chosen the projection we can now proceed to plot the map itself. This can be done either with simple contour lines or with filled contours. Here are two examples:
\[
\text{subplot(2,2,1); m_coast;}
\]
\[
\text{subplot(2,2,2); m_coast('patch',[.7 .7 .7]);}
\]
\[
\text{subplot(2,2,3); m_elev;}
\]
\[
\text{subplot(2,2,4); m_elev('contourf',[0:500:5000]);}
\]
After the map is drawn, we add grid and axes. In order to add a grid to all the four plots in the figure we can create a loop
\[
\text{for i=1:4; subplot(2,2,i); m_grid; end;}
\]
The result is shown in figure 21.

6.3 Plot data or add text to a map

Normally we have the data as functions of latitude and longitude. To project the cartesian grid to the map-grid use the function \text{m_ll2xy}. The inverse function, converting the data from the map-grid to a cartesian grid, is called (of course) \text{m_xy2ll}. Here is an example were we use a satellite projection and add text to position 5.33°, 60.4°N:
\[
\text{clf; m_proj('Satellite',} \text{lon},0,\text{lat',60});
\]
\[
\text{m_coast('patch',[.7 .7 .7]); m_grid;}
\]
\[
[x,y]=m_ll2xy(5.33,60.4); \text{text(x,y,'Bergen');}
\]
The result is shown in figure 22a. Here is another example where we plot global mean surface temperatures (satellite based) for January 1999 on such a map. We read the data with
\[
\text{fid=fopen('/home2/tore/Mat/sst1999jan.bin',} \text{'r'});}
\]
\[
\text{data=fread(fid,} \text{'float32'});}
\]
\[
\text{fclose(fid);}
\]
\[
[X,Y]=meshgrid(-179.5:179.5,-89.5:89.5);
\]
\[
\text{data=reshape(data,180,360);}
\]
These data are stored in the usual latitude-longitude format. Both the transformation to the chosen projection and the contour plot can be done with the function \text{m_contourf}:
\[
\text{m_contourf(X,Y,data,[-2:2:30]); m_coast('patch',[.7 .7 .7]);}
\]
Figure 21: Demonstration of M_map (section 6.1 and 6.2).

```
m_grid; colorbar;
```

The result is shown in figure 22b. The dateline may cause some problems since the position matrix goes from 179.5W to 179.5E. A 'pie' with a width of one degree is therefore missing. We could have fixed this by typing

```
X(:,361)=X(:,1); Y(:,361)=Y(:,1); data(:,361)=data(:,1);
```

As a last example we will demonstrate a more advanced plot. It uses 6 maps in the same figure window giving the same effect as when you peel an orange: The Earth is unfolded. The dateline also causes some problems here. I therefore had to make an extra dataset with longitudes from 0 to 360 which was plotted in the Pacific Ocean. See the result in figure 23.

```
Slongs=[-100 43::75 20 145;43 100;145 295;100 295];
Slat=[ 0 90;-90 0; 90;-90 0; 0 90];
X2=X(:,[181:360 1:180]); X2(:,[181:360])=X2(:,[181:360])+360;
Y2=Y(:,[181:360 1:180]); data2=data(:,[181:360 1:180]);
for l=1:6,
    m.proj('mollweide', 'long', Slongs(:,l), 'lat', Slat(:,l));
```
Figure 22: Demonstration of the satellite projection in M_map (section 6.3). To the right is SST (sea surface temperature) data from the file /home2/tore/Mat/sst1999jan.bin, and the mean temperatures for January 1999 are displayed.

```matlab
if l<5 m_contourf(X,Y,data,[-2:2:30]); hold on;
else m_contourf(X2,Y2,data2,[-2:2:30]); hold on; end;
m_grid(['fontsize',6,'xticklabels',[],'xtick',[-180:30:180],... 'ytick',[-80:20:80], 'yticklabels',[],'linest','-','color','k');
m_coast('patch',[.6 .6 .6]);
end;
set(gca,'ximmode','auto','ylimmode','auto');
```

6.4 Coastal data

Together with the map plotting routines are also high-resolution coastline data (typically 10-100 meter) and topography data gridded at intervals of 5 minutes (5-10 km resolution). The coastline data are “A Global Self-consistent, Hierarchical, High-resolution Shoreline Database (GSHHS)” that can freely be downloaded from the NOAA National Data Center (http://www.ngdc.noaa.gov/mgg/shorelines/gshhs.html). Responsible for the database is the emigrated norwegian Dr. Paul Wessel, SOEST, University of Hawaii, Honlulu (wessel@soest.hawaii.edu). These data can be used with different resolutions.

We will show a few examples of how to use the data for our coastline. All datasets used are global.

```matlab
m_proj('Lambert Conformal Conic','lon',[4 6],lat',[60 61]);
cf; m_coast('patch',[.7 .7 .7]); m_grid; print -deps kysta.eps
```
Figure 23: Demonstration of the Mollweide projection in M_map (section 6.3). The temperatures from the file /home2/tore/Map/sst1999jan.bin are added to the map.

```matlab
≫ clf; m_gshhs_c('patch', [.7 .7 .7]); m_grid; print -deps kystb.eps
≫ clf; m_gshhs_f('patch', [.7 .7 .7]); m_grid; print -deps kystc.eps
≫ clf; m_gshhs_j('patch', [.7 .7 .7]); m_grid; print -deps kystd.eps
≫ clf; m_gshhs_h('patch', [.7 .7 .7]); m_grid; print -deps kyste.eps
≫ clf; m_gshhs_f('patch', [.7 .7 .7]); m_grid; print -deps kystf.eps
```

The 6 plots are shown in figure 24. The files with the high-resolution coastal data are very large and therefore slow to read. With the function `m_gshhs_f` it took me about 100 seconds. This is far too long if you are going to plot many times. What you can do is to store the data, needed to plot the map, in a smaller file. This can be done with `m_gshhs_f('save', 'filename')`, which can also be used for the other `m_gshhs` routines. Having stored the data you can plot them with `m_usercoast('filename', ...)`. Here is an example where we first save the map data in the file `min_coast`, and then plot the map.

```matlab
≫ m_gshhs_f('save', 'min_coast');
≫ m_usercoast('min_coast', 'patch', [.7 .7 .7]);
```

By comparison this time we needed only 5 seconds to plot the data. The efficiency of this method should be obvious when the file used by `m_gshhs_f` is 89 MB compared to `min_coast` of only 0.1 MB.

6.5 Topography

`M_map` contains two different datasets; one with 5 minute resolution and one with 1 degree resolution. The high-resolution set is the global ETOPO5, also freely available from the NOAA Na-
Figure 24: Demonstration of the different coastline datasets in M_map (section 6.4).
Figure 25: Demonstration of the topography data in M_map (section 6.5).

tional Data Center (http://www.ngdc.noaa.gov/mgg/global/geltopo.html). The coarse dataset is created from the ETOPO5 set, using the mean of the 1x1 grid boxes.

Here is a map of Europe where we use the topography data from the low-resolution database,

\[
\texttt{figure(1); clf; m_proj('lambert', 'long', [-30 50], 'lat', [30 80]);}
\]
\[
\texttt{m_elev('contourf', [-6000:500:6000]); m_coast('Linewidth', 2);}
\]
\[
\texttt{m_grid('box', 'fancy', 'tickdir', 'in');}
\]
\[
\texttt{xlabel('Map of Europe', 'visible', 'on');}
\]

together with a map of the Nordic Seas using the high-resolution data.

\[
\texttt{figure(2); clf; m_proj('lambert', 'long', [-15 15], 'lat', [60 70]);}
\]
\[
\texttt{m_base('contourf', [-6000:500:6000]); m_gshhs('Linewidth', 2);}
\]
\[
\texttt{m_grid('box', 'fancy', 'tickdir', 'in');}
\]
\[
\texttt{xlabel('Kart over Norskehavet', 'visible', 'on');}
\]

The two maps are shown in figure 25.

6.6 Exercises to section 6

We will now practice plot and map functions in Matlab. For this purpose we will use the files /home2/tore/Mat/salt.mat and /home2/tore/Mat/temp.mat consisting of observed salinity and monthly mean temperatures from a depth of 100m in the Norwegian Sea. The data are in a 1 x 0.5° grid, and the position of every grid cell is also stored in the files.

1. Read the temperature data and find the yearly mean. Name this variable tempm, and draw a testplot using the contourf function.
2. Choose the projection *Lambert Conformal Conic* on the area *lon* from 32W to 22E and *lat* from 59N to 83N. Plot the data with color contours at intervals of 0.5°C. Remove the contour lines. (*Hint: Get the object number from contour, i.e. [cs,h]=contourf..., and then look at set(h(1),’Edgecolor’)).

3. Add land topography with intermediate resolution. Add a grid. You will discover that the colors disappear. This is an inconsistency between *Matlab* and *m_map*. If you save the figure in postscript format and view it in *ghostview*, you will see that everything is ok.

4. Repeat the same procedure for salinity. Make sensible color intervals.

5. Can the two plots tell you anything about the ocean currents in the Norwegian Sea? Create a plot of the topography in the Norwegian Sea by using the *m_tbase* function. Is there a relation between topography and current?
7 Scripts and functions: Create your own Matlab routines

In this section we will show you how to create your own scripts and functions, and store them in files available to the Matlab Command Window. Such files must have an .m extension, ie my_super_program.m. All functions and scripts in Matlab have filenames of this form. The paths can be found with the function which. Try which mean or which demo.

From the Matlab Command Window you can run these so-called M_files if they are placed in the same directory as you started Matlab from or in a directory under your home directory named matlab. Alternatively, you can set the path to any directory with the function path, which in Unix should be of the form
g path(path, '/home2/tobald/m_filer')
and in DOS of the form
g path(path, 'Data\matlab\mfiler')

This can be done automatically when you start Matlab by creating a file called startup.m placed in a matlab directory under your home directory. If Matlab finds this file the commands are executed every time you start Matlab. An example of such a file is

% startup, adds paths to own Matlab routines.

path(path,'/home2/tore/matlab/eigne_filer')
path(path,'/home2/tore/matlab/SeaWater')
path(path,'/home2/tore/matlab/Couple')

% files to netcdf-library:
toolbox_area = '/dataRRTF/tore/ Matlab/Netcdf/';

path(path, fullfile(toolbox_area, 'netcdf', ''))
path(path, fullfile(toolbox_area, 'netcdf', 'nc_type', ''))
path(path, fullfile(toolbox_area, 'netcdf', 'nc_utility', ''))
path(path, fullfile(toolbox_area, 'netcdf', 'nc_files', ''))

7.1 Matlab scripts

A Matlab script is a collection of commands stored in a file with name name.m. This file can be executed in the Command Window by just typing name. All variables used in the script are stored in the Matlab memory. As mentioned in section 2.6 on page 8 the command lookfor text name will search for the text text in the first comment-line of the file, help name will print all the first connected comment-lines, and type will print the entire file. You should therefore have essential information at the beginning of the script so that it is available to the help and lookfor functions.

To demonstrate the structure of a typical script, we create a program that reads data from the file ctd.dat, plots a temperature-salinity graph and, on the same graph, plots a map showing where the data are from. The commands used to read the data are taken from section 4.2.2, the
plotting from section 4.1.1 and mapmaking from section 6.5. The command lines shown below are stored in the file ctd1.m in our matlab directory.

% Program ctd1
% 
% Made 14/3/2000 by Tore Purevik
% 
% Reads data from ctd1.dat, plots a ts-diagram, og shows the % position of the station on a map.

% Read data
fid=fopen('/home2/tore/Mat/ctd1.dat','r');
dummy=fread(fid,'char');
fclose(fid);

% Find position to the information we want from the header
c=char(dummy'); date_index=findstr('Date:',c);
time_index=findstr('St.Time',c); lon_index=findstr('St.Lon',c);
lat_index=findstr('St.Lat',c);

% Store the information in variables
date=c(date_index+6:date_index+16);
time=c(time_index+9:time_index+16);
lon=c(lon_index+8:lon_index+20);
lat=c(lat_index+8:lat_index+20);
d=str2num([lon(1:3) ' ' lon(6:7) ' ' lon(9:13)]); lon=d(1)+d(2)/60+d(3)/3600;
d=str2num([lat(1:3) ' ' lat(6:7) ' ' lat(9:13)]); lat=d(1)+d(2)/60+d(3)/3600;

% Get the measurements data
nobs_index=findstr('N OBS',c);
ctd=str2num(c(nobs_index+5:end));

% Plot temperature vs salt
plot(ctd(:,2),ctd(:,4)); grid; xlabel('Salt (psu)');
ylabel('Temperature (C)'); axes([34.65 35.15 -2 7]);
set(gca,'XTick',[34.65:.05:35.15],'YTick',[-2:7]); merke=[10:10:50 100:100:500 1000 2000]; hold
on;h=plot(ctd(mark,2),ctd(mark,4),'.'); hold off;
seth('Marksizer',14);
h=text(ctd(mark,2)+.005,ctd(mark,4),num2str(mark)); title(['Ctd measurements ' date', kl ' time])

% Plot map in a new graph at the bottom left hand corner

64
Figure 26: The result of the script ctd1. The data are from the file /home2/tore/Mat/ctd1.dat.

pos=get(gca,'Position');
axes('Position',[pos(1)+.05 pos(2)+.05 .3 .3]);
m_proj('lambert','long',[-20 20],'lat',[74 81]);
m_elev('contour',[-3500:1000:-500],'edgecolor','b');
m_coast('patch',[.7 .7 .7]);
m_grid('box','fancy','tickdir','in');
[x,y]=m_l12xy(lon,lat); h=plot(x,y,'r'); set(h,'Markersize',14);

% Print to file.
print -depsc ctd1.epsc;

Then we remove all variables from memory with clear all, and type ctd1. It takes approx. 2
seconds to read and plot the data. The result is shown in figure 26. Note that if we type help
ctd1, the first 6 lines are shown. If we type whos we can see that all the variables now are in the
Matlab memory.

If we want to repeat the same operation with many files, we can use a function called eval. It
executes commands given as strings in the input arguments. Here is an example where we read
data from 3 CTD files, plot them in figure windows numbered 1 to 3, and save the postscript
figures as ctd1.epsc, ctd2.epsc and ctd3.epsc. The only thing we need to change in the program
ctd1.m are two lines at the start and end of the function. Here is the new program called
ctd_all.m:

% Program ctd_all


65
Figure 27: The result of ctd_all. The data are from the files /home2/tore/Mat/ctd2.dat and /home2/tore/Mat/ctd3.dat.

% Made 14/3/2000 by Tore Furevik
% % Reads data from ctd1.dat, ctd2.dat and ctd3.dat, plots the data in a
% % ts-diagram, shows the position of the station on a map, and prints to the
% % files ctd1.epsc, ctd2.epsc and ctd3.epsc

% Lese inn data
for i=1:3,
    figure(i); clf;
    fid=fopen([’/home2/tore/Mat/ctd’ num2str(i) ’.dat’],’r’);
    : (as ctd1.m)
    :
    eval([’print -depsc ctd’ num2str(i) ’.epsc’]);
end;

In addition to the plot in figure 26, we now also create the plots in figure 27.

7.2 Matlab functions

In section 2.6 on page 8 we printed the function mean as an example of a Matlab function. We will here briefly discuss the commands used in the mean function:

function y = mean(x,dim)
if nargin==1,
% Determine which dimension SUM will use
    dim = min(find(size(x) == 1));
    if isempty(dim), dim = 1; end
    y = sum(x)/size(x,dim);
else
    y = sum(x,dim)/size(x,dim);
end

A clever feature in Matlab is that although a function takes two input-arguments (in this case the matrix x and the variable dim), it can also be used with just one input. This makes the functions very flexible. The function nargout checks the number of input-arguments. If there is only one, the function decides along which dimension the mean value should be calculated. If, for instance, x is given by x=[2 3; 1 2; 3 4], the command find(size(x) == 1) will return [1 2], that is, we have a MxN matrix where both M and N are larger than 1. In such cases the function mean calculates the mean value of the columns by using the functions sum and size columnwise. If there are two input-arguments the mean value is taken along the dimension given in argument 2. That is mean (x,1) returns [2 3] and mean(x,2) returns [2.5; 1.5; 3.5].

Now we will show how we can transform the script ctd1 into a function where we can choose the output to be one or more of the variables ctd, lon, lat, date and time, or to plot the data in the same way as before. As input-argument we use a number representing the file we want to load. The function will look like this

function [ut1,ut2,ut3,ut4,ut5]=ctd_one(filename);
    % Function [ctd,lon,lat,date,time]=ctd_one(filename);
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Made 14/3/2000 by Tore Furevik
    %
    % Reads data from a ctd file (standard er 1).
    % Input argument: filename - number of the file to be plotted (between 1 and 3).
    % Output argument: ut1 (ctd) - matrix containing pressure, salinity, density,
    % temperature and number of observations (flyttal)
    % lon - longitude of the ctd-station (flyttal)
    % lat - latitude of the ctd-station (flyttal)
    % date - date of the ctd-station (streng)
    % time - time of the ctd-station (streng)
    %
    % check input arguments
    if nargin==0, disp('Filename set to 1'); i=1; end; if nargin==1,
       if isstr(filename),
       disp('The file number must be an integer'); return;
       else i=filename; end;
       if i<1,
       disp('The file number must be a positive integer. Value 1 is chosen'); i=1;
       elseif i>3,
       disp('There are at the moment only 3 files. Value 3 is chosen'); i=3;
       %
       end}

67
end;
end;
if nargin>1,
    disp('You can only give one file number'); return
end;

% Reading the data
    : (as ctd_all.m)
    :
if nargout<1, % insätt i linja over ' % Plotte temperatur mot salt'
    : (as ctd_all.m)
    :
if nargout>0, ut1=ctd; ut2=lon; ut3=lat; ut4=date; ut5=time; end; % last line

We then store it as ctd_one.m, remove all variables from memory with clear all, and test our function

≫ ctd_one(0);
and Matlab responds with

The file number must be a positive integer. Value 1 is chosen
and plots the data in ctd1.dat in figure window number 1. Type whos to convince yourself that no variables are stored in memory.

If we accidently type

≫ ctd_one('d');
Matlab responds with

The file number must be an integer
and nothing else happens.

Now we want the data from the file ctd2.dat, and we type

≫ [data,length,width,date,kl]=ctd_one(2);
The command whos displays the following list of variables:

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>1x1</td>
<td>8</td>
<td>double array</td>
</tr>
<tr>
<td>data</td>
<td>2604x5</td>
<td>104160</td>
<td>double array</td>
</tr>
<tr>
<td>date</td>
<td>1x11</td>
<td>22</td>
<td>char array</td>
</tr>
<tr>
<td>kl</td>
<td>1x8</td>
<td>16</td>
<td>char array</td>
</tr>
<tr>
<td>length</td>
<td>1x1</td>
<td>8</td>
<td>double array</td>
</tr>
</tbody>
</table>

Grand total is 13041 elements using 104214 bytes

68
Type \texttt{kl} to see whether this variable really is the time. Then in the end, if we want to extract the ctd data from the file \texttt{ctd1.dat}, we can do this with the command

\begin{verbatim}
\tt>> data=ctd_one(1);
\end{verbatim}

and if we want the data, latitude and longitude, we can type

\begin{verbatim}
\tt>> [data,lat,lon]=ctd_one(1);
\end{verbatim}

This was a brief introduction to Matlab functions. When you learn to use this powerful tool you are heading for a bright future!

\section*{7.3 Exercises to section 7}

We will now practice a few simple functions and scripts. Here are some exercises:

1. In your home directory create a directory called \texttt{matlab} (you may have such a directory already). And then under \texttt{matlab} create a directory called \texttt{functions}. Then add the path to this directory in a \texttt{startup.m} file in your \texttt{matlab} directory. If your home directory, for instance, is \texttt{teobald} and located on \texttt{/home2}, the \texttt{startup.m} file should include a line like \texttt{path(path,'/home2/teobald/matlab/funksjoner')}.

2. Open a texteditor (\texttt{vi}, \texttt{xemacs}, \texttt{edit}) and create a function called \texttt{stat} and save it in the \texttt{functions} directory. The function shall have a n-dimensional matrix \texttt{X} as input and return an overview of the matrix. For example \texttt{stat(rand(3,2,3,9,12,2))}, producing the information

\begin{verbatim}
The matrix has the size: \texttt{3 2 3 9 12 2}  
Minimum is: \texttt{0.00039022}  
Maximum is: \texttt{0.9998}  
Mean value is: \texttt{0.50468}  
Standard deviation is: \texttt{0.28915}
\end{verbatim}

3. Edit the function \texttt{stat} so that you can have an output argument as well. If you type \texttt{stat(X)} the same information will be displayed on the screen as before, but if you for instance type \texttt{s=stat(X)}; the matrix information will be stored in the variable \texttt{s} (\texttt{hint: Use a nargout check}).

4. A good way of presenting distribution of data is the \texttt{boxplot} function. It creates a boxplot showing the mean value as a horizontal line, the first and third quartiles as the lower and upper limit of the box, and with dashed lines to the minimum and the maximum values. Add an option in your \texttt{stat} function, by including a parameter as input argument, to do a \texttt{boxplot}. The commands \texttt{stat(X)} or \texttt{stat(X,0)} will yield the same results as before, while \texttt{stat(X,1)} will also visualize the data in a \texttt{boxplot}.

\begin{verbatim}
\texttt{Correspondence: 69}
\end{verbatim}
Tore Furevik, The Geophysical Institute, University of Bergen Allégaten 70, N-5007 Bergen.
Tel: 47 55582691, email: tore@gfi.uib.no, nett: www.gfi.uib.no/~tore