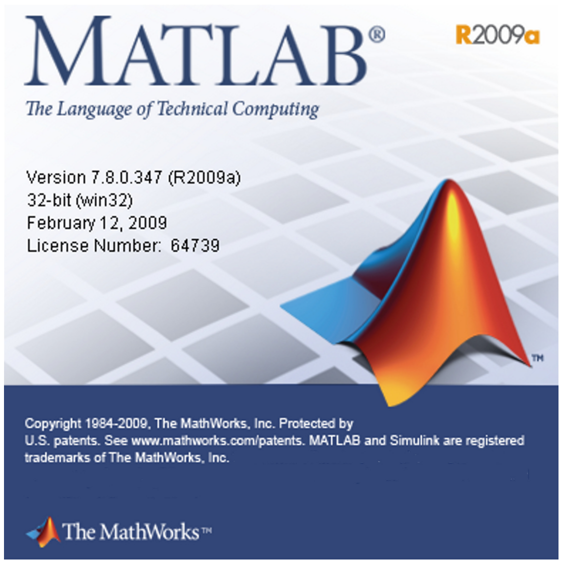
# Programming in MATLAB


MATLAB® R2009a

The Language of Technical Computing

Version 7.8.0.347 (R2009a)
32-bit (win32)
February 12, 2009
License Number: 64739

Copyright 1984-2009, The MathWorks, Inc. Protected by U.S. patents. See www.mathworks.com/patents. MATLAB and Simulink are registered trademarks of The MathWorks, Inc.

The MathWorks™

Universiteit van Amsterdam

February 23, 2010

Prof. Dr. Ir. W. Bouten
Peter Kraal
Jurriaan H. Spaaks

# Contents

# List of Projects

# List of Tables

# Chapter 1

# Introduction

When you watch the weather forecast, all predictions are based on the results of computer models. With real-time input of meteorological conditions and a model that simulates (part of) the global climate system, it is possible to predict the weather for the near future. Of course these models can never be 100 percent accurate, as everybody who has gone out in shorts and ended up soaked with rain can confirm.

In another example of modeling, the United Nations use computer models to predict the development of global phenomena like diseases, famine and death. Other model simulations based on demographics and population data can be used to estimate future trends in world population and migration.

Science has made tremendous progress in the last decades. Measuring techniques, instruments and knowledge transfer are becoming better and more efficient. But the examples above show the importance of another factor: the increasing use of computers to analyze and generate data. It is now possible to quickly access, analyze and manipulate amounts of data on your computer that would have taken up complete libraries in the past. And the possibilities reach beyond simply using existing data: models can approximate real-life processes and generate data that was not available before. Furthermore, models can deepen the understanding of the earth and all of the processes influencing our environment.

The computer programs you may have so far encountered in your life - Microsoft Excel, Word and PowerPoint, ArcGIS and Stella - are all packages for specific tasks. Excel, ArcGIS and Stella allow you to analyze, manipulate and visualize data to some degree. However, you are always restricted to the features that the software developers included in these packages.

Alternatively, programming languages, such as Fortran, Basic, C, C++, Pascal, Turbo Pascal, Object Pascal (Delphi), R, Python and MATLAB give you the freedom to write your own programs that simulate and calculate almost anything you want them to. Many specific scientific problems cannot be tackled without a custom-made computer program.

In this course you will use MATLAB, a 'next-generation' programming language based on C and Fortran, but without all the programming overhead that comes with those languages, making MATLAB much easier to use in comparison.

At the University of Amsterdam the generation and transfer of programming knowledge has become an important feature of the educational program. It is the believe—and rightly so—that a student who is able to write his own computer programs, and understand those of others, has more flexibility when analyzing, manipulating and understanding data. With Excel you cannot realistically model global climate change or any other (complicated) earth process. At this moment, MATLAB is being used to model hydrological systems, erosion in the semi-arid Mediterranean, rockfall in the Alps, the interaction between soil and water, or bird migration from Scandinavia to Africa, to name but a few current research topics where programming is involved.

MATLAB's user-friendly interface and programming flexibility have made it a popular tool for scientists. On the Internet, large amounts of MATLAB programs and tutorials can be found. Furthermore, MATLAB is an 'array-oriented' programming language and therefore very suitable for spatial (1D, 2D, 3D, 4D. . . ) calculations.

Because of all this, we think it is important that a student in Earth Sciences at the University of Amsterdam be introduced to MATLAB at an early stage. This course, Programming in MATLAB, has been developed to make you comfortable with programming in MATLAB and teach you how the Earth's processes can be converted into MATLAB programs.

**Take your time going through the exercises. Make sure you try to interpret your results. If you rush too much, you won't be able to pick up the information and you will easily forget what you have learned.**

# Chapter 2

# Getting started

- ▶ Start MATLAB by clicking “Start” ➙ “Programs” ➙ “MATLAB” ➙ “MATLAB R2009a” and click on “MATLAB R2009a”.
- ▶ Go to “Desktop” in the menu bar and select “Desktop Layout” and then choose “Command Window Only”.

The Command Window is the window where you type your commands at the ‘>>’-sign, also known as ‘prompt’. These commands are then being executed by MATLAB after pressing the “Enter” key.

## 2.1 Creating your own work directory

To create your own work directory, click the “Browse for Folder” button (see Figure 2.1) located to the right of the “Current Directory” drop-down list on the MATLAB button bar. This button is used to change or create your work directory. Click to create your work directory or to change the directory to the directory you will be working from. Now go to the “Desktop” menu and click a checkmark next to “Current Directory”. You will see that you are now in the directory you selected through using the “Browse for Folder” button.

**TIP:** Make it a habit to start every new exercise by selecting the appropriate directory for that exercise. Be sure that you are in that directory when executing your commands.

- ▶ Type

```
>> Groningen = 2.33
```

and then press “Enter”, keeping in mind that ‘>>’ is the command prompt. Note that the MATLAB Command Window now displays the following output:

```
Groningen =

    2.3300
```

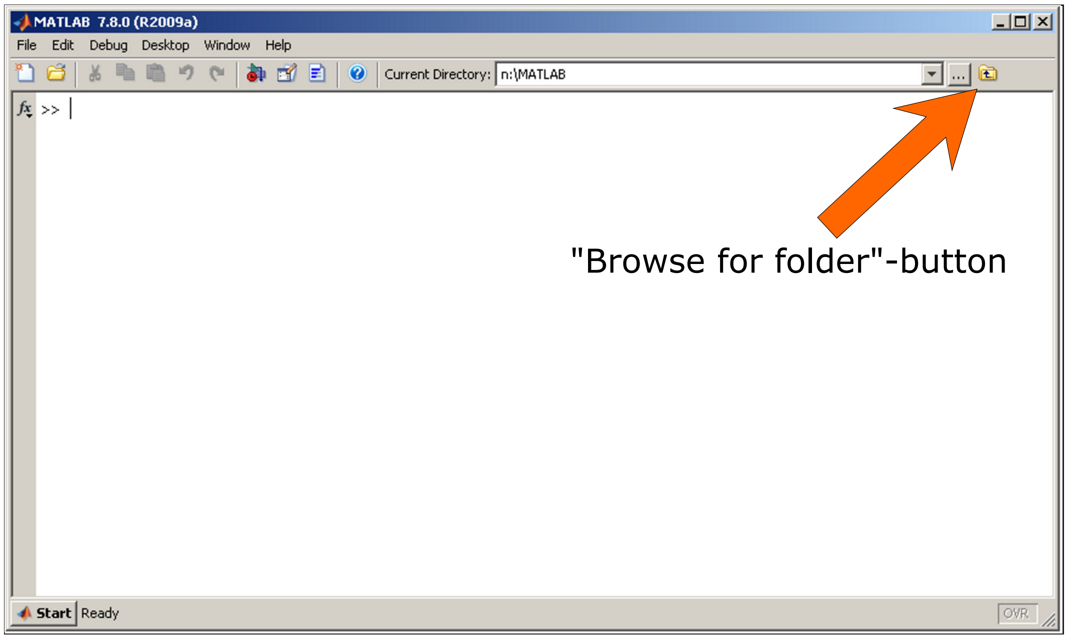

MATLAB 7.8.0 (R2009a)
File Edit Debug Desktop Window Help
Current Directory: n:\MATLAB
fx >>
"Browse for folder"-button
Start Ready
OVR

Figure 2.1: Location of the “browse for folder”-button.

This output lets you know that MATLAB has created a variable named `Groningen` whose value is equal to `2.3300`.

Go to the “Desktop” menu and click a checkmark next to “Workspace”, and you will see the `Groningen` variable listed in the MATLAB Workspace. Alternatively, you can type `who` or `whos` at the command prompt. These commands will list the variables in your Workspace in the Command Window.

▶ Type

```
>> Utrecht = 2.12
```

and then press “Enter”.

▶ Type

```
>> Maastricht = 1.92
```

and then press “Enter”. Both the `Utrecht` and `Maastricht` variables are now listed in the MATLAB Workspace window.

▶ Calculate the mean of the total yearly precipitation for these three cities:

```
>> Prec = (Groningen + Utrecht + Maastricht)*365/3
```

Now, not only do `Groningen`, `Utrecht`, and `Maastricht` appear in the workspace, but a new variable `Prec` is also present. These short calculations are simple examples of the arithmetic operations that can be executed in MATLAB. Table 2.1 gives an overview of the basic mathematic operators used in MATLAB. Note: The square root is merely an instance of the Exponentiation notation: `A^0.5`≡`sqrt(A)`.

▶ Clear the workspace using the command `clear`

Table 2.1: MATLAB operations.

| Operation | Symbol | Example |
|---|---|---|
| Addition | `+` | `5+6.7` |
| Subtraction | `-` | `8.63-5.24` |
| Multiplication | `*` | `3*2` |
| Division | `/` | `1/4` |
| Exponentiation | `^` | `3^2` |

▶ Assign the value 1 to the variable `ExampleVar`:

```
>> ExampleVar = 1
```

▶ Assign the value 14 to the variable `EXAMPLEVAR`:

```
>> EXAMPLEVAR = 14
```

▶ Check the workspace by typing:

```
>> whos
```

and then pressing the "Enter" key.

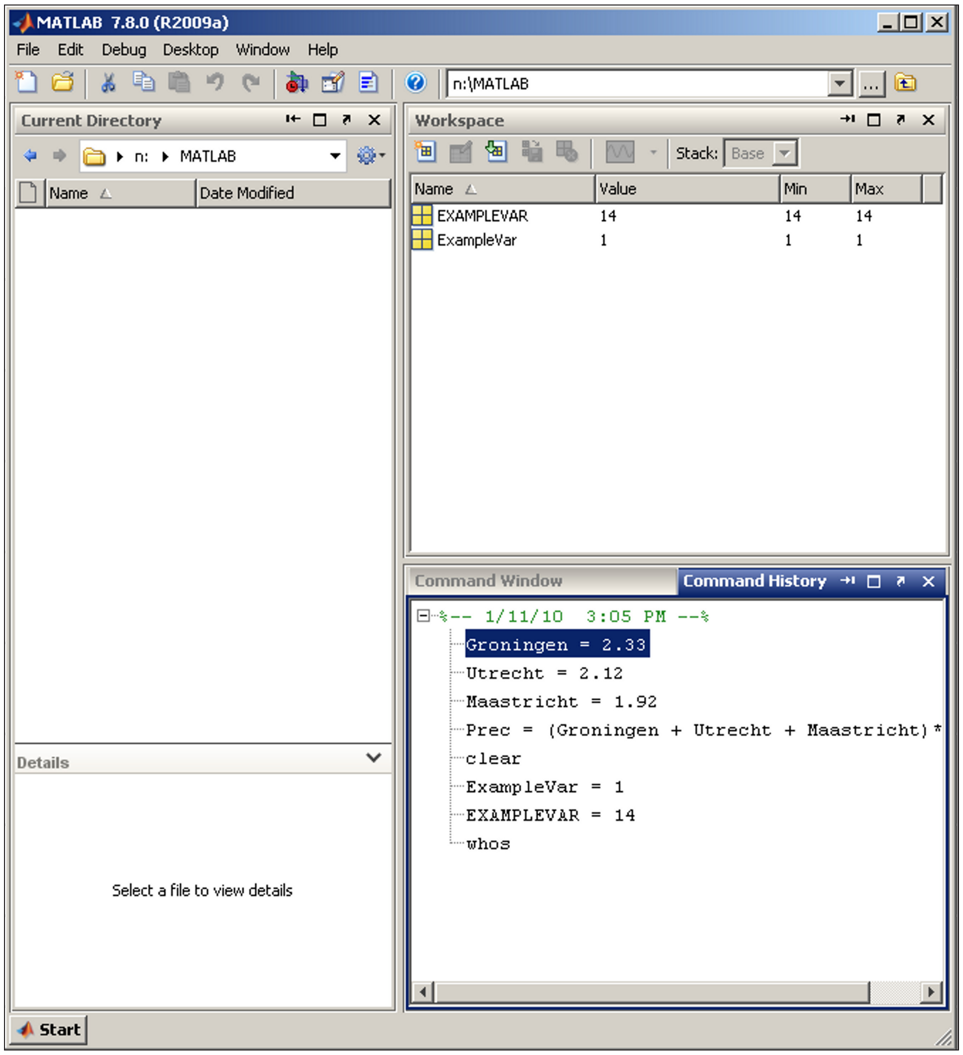
```
  Name               Size                       Bytes  Class          Attributes

  EXAMPLEVAR         1x1                            8  double array
  ExampleVar         1x1                            8  double array
```

This procedure is meant to demonstrate that MATLAB is case-sensitive: `EXAMPLEVAR` and `ExampleVar` are two different variables. Another way to check structures of variables entered into the MATLAB Workspace is to double-click the variable in the MATLAB "Workspace" window. This opens the variable up in the array editor. Check the value of the variable `EXAMPLEVAR` by double-clicking its variable in the "Workspace" window.

▶ Check the value of the variable `EXAMPLEVAR`.

**TIP:** Be aware that as we work through the manual, the use of any special text characters (such as `*`, `&`, @, `#`, and `?` ) within variable names is prohibited, the exception being the underscore '_' character. This is because these characters often have special effects on operations. Variable names can contain up to 31 characters, but must start with a letter followed by any combination of letters, digits or underscores.

▶ Now, go to the "Desktop" menu and click a checkmark next to "Command History". This opens a new window within the MATLAB Graphical User Interface (GUI) listing the commands that you have entered in the "Command Window". Figure 2.2 shows you what your GUI should look like now.



| Name | Value | Min | Max |
|---|---|---|---|
| EXAMPLEVAR | 14 | 14 | 14 |
| ExampleVar | 1 | 1 | 1 |

```
%-- 1/11/10  3:05 PM --%
Groningen = 2.33
Utrecht = 2.12
Maastricht = 1.92
Prec = (Groningen + Utrecht + Maastricht)*
clear
ExampleVar = 1
EXAMPLEVAR = 14
whos
```

Figure 2.2: MATLAB Graphical User Interface (GUI). Note that the Command Window and Command History are docked into the same panel. You can change the layout of your GUI by clicking the title bar of the panel that you want to put somewhere else, and moving the mouse pointer while holding down the left mouse button.

**TIP:** MATLAB remembers the commands that have been typed at the prompt. You can use the up and down arrows to select any previously typed commands. You can also check the "Command History" item in the "Desktop" menu.

## Chapter 3

# Arrays: scalars, vectors and matrices

Besides using MATLAB as a common calculator, it is possible to perform calculations on entire datasets. These variables must be stored in the workspace and addressed by their variable name. Variables are stored in the MATLAB workspace as arrays. The four main array types that we will use are:

**Scalar** 1x1 array containing one value

**Vector** a row or column array (1xM or Mx1 array) containing M values

**Matrix** a 2-dimensional array (e.g. a map)

**Multi-dimensional arrays** e.g. a 5x2x3 array containing 30 values

**TIP:** You can retrieve the size of an array by issuing a `size(X)` command, where `X` represents the array that you want to know the size of.

## 3.1 Data types and array classes

MATLAB arrays can contain three different data types: integer, floating point, and character data.

**TIP:** A character variable is often referred to as a ‘string’, especially when it contains more than one character.

A description of different data types and the array classes in which they are stored is given in Table 3.1.

Table 3.1: Array classes

| Data type | Examples | Description | Array class |
|---|---|---|---|
| Integer | `-2 -5 -8 0 2 34 221` | (+/-) Whole numbers | Double array |
| Floating point | `3.2232 8.323 5823.22` | (+/-) Fractional numbers | Double array |
| Character | `y L \ = ] - K , u ;` | ASCII text characters | Character array |

## 3.2 Creating arrays

Besides using existing datasets for analysis, it is important to be able to create and manipulate your own arrays within the MATLAB environment. This can vary from creating a simple vector with an increasing value from 1 to 365 to serve as a time indicator for a certain process to complex 4-D arrays with certain parameters at a place in space (X, Y, Z) that change over time. The value of each position in an array can then be manipulated to simulate a process. These operations can be performed on complete arrays or selected cells.

▶ Type

```
>> A = [1,2,3;4,5,6]
```

`A` is now present in the workspace as a 2x3 double array. When declaring arrays other than scalars, the array contents must be in brackets `[ ]`. If you take a closer look at the output that MATLAB returns, you'll see that you just made an array with 2 rows and 3 columns.

**TIP:** Commas (`,`) serve as column delimiters, and semicolons (`;`) are row delimiters.

There are several ways to create vectors:

▶ Type

```
>> k = [1:1:9]
```

press "Enter" and interpret the result.

▶ Type

```
>> m = [21:-2:13]
```

press "Enter" and interpret the result. As you can see, the syntax of this command is `[minimum:increment:maximum]`.

▶ Generate an array starting with 10.2, decreasing with 0.3 and ending at 6.0.

▶ As simple as you can generate 1-D arrays, you can also make 2-D arrays; type

```
>> F = [-10:1:10;40:-1:20]
```

and interpret the result.

- ▶ Type

  ```
  >> G = [-5:1:15;8:-0.5:1.0]
  ```

  and interpret the error.

- ▶ Once an array contains one of the three data types, it cannot contain another data type as well. Type

  ```
  >> H = [1,2,3;'PiM']
  ```

  and interpret the result.

- ▶ Now type

  ```
  >> I = [1,2,3,4;'PiM']
  ```

  and interpret the error.

## 3.3 Manipulating arrays by their subscripts and indices

Elements of arrays are identified by their positions in the array. So...

```
>> F(2,6)
```

returns 35. This is the value of the element in the second row and the sixth column in matrix `F`.

Elements can also be identified by their index. The index is a counter that starts at the first row and first column, then continues down in the first column and then takes next column. To clarify the difference between indices and subscripts, take a closer look at the tables below:

Table 3.2: Array elements identified by subscripts

| | | |
|---|---|---|
| (1,1) | (1,2) | (1,3) |
| (2,1) | (2,2) | (2,3) |
| (3,1) | (3,2) | (3,3) |

Table 3.3: Array elements identified by indices

| | | |
|---|---|---|
| (1) | (4) | (7) |
| (2) | (5) | (8) |
| (3) | (6) | (9) |

- ▶ Retrieve the value of the element denoted by subscript 2,1 in matrix `A`. What value do you expect?

  ```
  >> A(2,1)
  ```

---

**TIP:** Parenthesis symbols `( )` are used to point to elements of an array, in contrast to the `[ ]` square brackets that are used to assign values to the array.

---

▶ What is the index of the value 23 in the matrix `F`? And what are the subscripts?

▶ Alter the values in a matrix by calling their indices or subscripts and assigning a new value:

```
>> F(20) = -40
```

Now check what has changed in the matrix `F`.

You have now altered one array element of matrix `F` by using a scalar as index. You can also change more than one array element of `F`, at the same time, by using an index vector instead of just one index scalar.

▶ Carefully interpret the sentence above and (try to) change the array elements of `F` to 50 where the index is equal to one of the elements of the vector `IndVec`, when given:

```
>> IndVec = [3,15,2,8,19]
```

▶ Can you this do this in a single command?

## 3.4 Matlab demonstrations

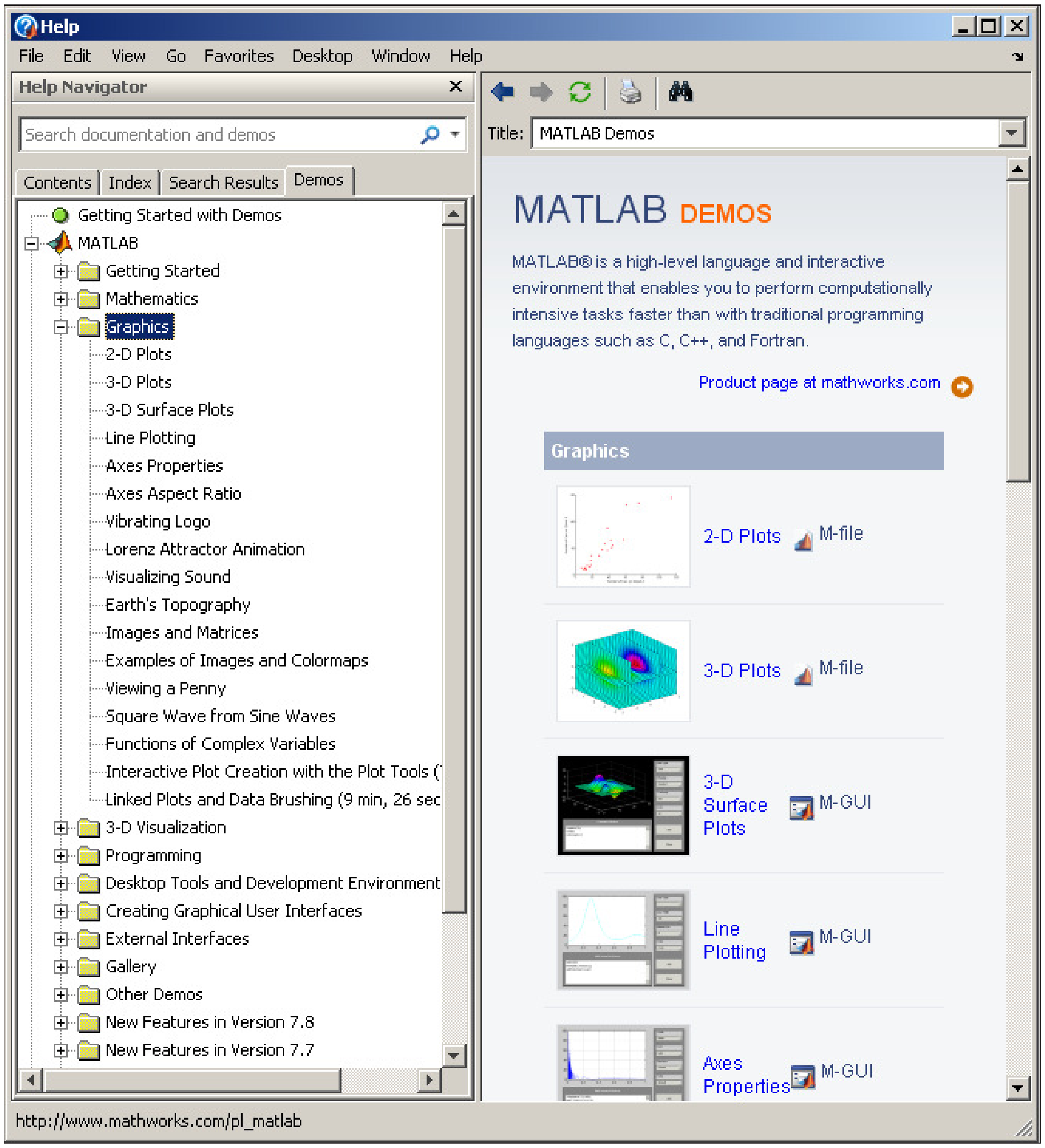
Maybe this is a good moment to take a break for a while and to have a look at some demonstrations for 10–15 minutes, as MATLAB has an extensive '`demo`' function. When you see what you can do with MATLAB, you have a better reference while studying this text and while doing the exercises. Select one or two of the most instructive demos and share these with your neighbors.

▶ Start MATLAB demonstrations by typing `demo` on the MATLAB command line.

The help documentation window that pops up consists of two panes. Read the contents of the right pane. After that, try expanding one of the menu tree items in the left pane, for example MATLAB ➙ graphics (see Figure 3.1).

## 3.5 Arithmetic operations on matrices and vectors

Just as you can perform calculations on scalars (as we did in the exercise with the precipitation), you could perform calculations on larger arrays as well. Some examples are outlined below:


Help
File Edit View Go Favorites Desktop Window Help
Help Navigator
Search documentation and demos
Contents Index Search Results Demos
Getting Started with Demos
MATLAB
Getting Started
Mathematics
Graphics
2-D Plots
3-D Plots
3-D Surface Plots
Line Plotting
Axes Properties
Axes Aspect Ratio
Vibrating Logo
Lorenz Attractor Animation
Visualizing Sound
Earth's Topography
Images and Matrices
Examples of Images and Colormaps
Viewing a Penny
Square Wave from Sine Waves
Functions of Complex Variables
Interactive Plot Creation with the Plot Tools (
Linked Plots and Data Brushing (9 min, 26 sec
3-D Visualization
Programming
Desktop Tools and Development Environment
Creating Graphical User Interfaces
External Interfaces
Gallery
Other Demos
New Features in Version 7.8
New Features in Version 7.7
Title: MATLAB Demos
MATLAB DEMOS
Product page at mathworks.com
Graphics
2-D Plots M-file
3-D Plots M-file
3-D Surface Plots M-GUI
Line Plotting M-GUI
Axes Properties M-GUI
http://www.mathworks.com/pl_matlab

MATLAB® is a high-level language and interactive environment that enables you to perform computationally intensive tasks faster than with traditional programming languages such as C, C++, and Fortran.

Figure 3.1: MATLAB demonstrations.

### 3.5.1 Multiplication of a scalar with a 2-D array

$$5 * \begin{bmatrix} 4 & 5 & 2 \\ 1 & 9 & 0 \\ 1 & 5 & 5 \end{bmatrix} = \begin{bmatrix} 20 & 25 & 10 \\ 5 & 45 & 0 \\ 5 & 25 & 25 \end{bmatrix} \tag{3.1}$$

Or, in MATLAB:

```
>> A = 5
>> B = [4,5,2;1,9,0;1,5,5]
>> C = A * B
```

### 3.5.2 Addition of 2-D arrays

$$\begin{bmatrix} -1 & 2 & 0 \\ 2 & -3 & 1 \\ -4 & 0 & 5 \end{bmatrix} + \begin{bmatrix} 4 & 5 & 2 \\ 1 & 9 & 0 \\ 1 & 5 & 5 \end{bmatrix} = \begin{bmatrix} 3 & 7 & 2 \\ 3 & 6 & 1 \\ -3 & 5 & 10 \end{bmatrix} \tag{3.2}$$

Or, in MATLAB:

```
>> D = [-1,2,0;2,-3,1;-4,0,5]
>> B = [4,5,2;1,9,0;1,5,5]
>> E = D + B
```

### 3.5.3 Multiplication of 2-D arrays ('dot-product')

$$\begin{bmatrix} 3 & 1 & 7 \\ 8 & 4 & 6 \end{bmatrix} \bullet \begin{bmatrix} 4 & 5 \\ 2 & 1 \\ 9 & 0 \end{bmatrix} = \begin{bmatrix} 77 & 16 \\ 94 & 44 \end{bmatrix} \tag{3.3}$$

Or, in MATLAB:

```
>> F = [3,1,7;8,4,6]
>> G = [4,5;2,1;9,0]
>> H = F * G
```

In general, elements of `H` are calculated using:

$$H(r,c) = \sum_{k=1}^{nElems} F(r,k) \times G(k,c) \tag{3.4}$$

in which $r$ is the row number, $c$ is the column number, $nElems$ is the number of columns in $F$ (or, equivalently, the number of rows in $G$). For example:

$$\begin{array}{ccccccccccccc} H(1,1) & = & F(1,1) & * & G(1,1) & + & F(1,2) & * & G(2,1) & + & F(1,3) & * & G(3,1) & = \\ & & 3 & * & 4 & + & 1 & * & 2 & + & 7 & * & 9 & = 77 \end{array}$$

Note that `F * G` returns a different result than `G * F`.

### 3.5.4 Multiplication of 2-D arrays (element-by-element)

$$\begin{bmatrix} 3 & 1 & 7 \\ 8 & 4 & 6 \\ 5 & 3 & 4 \end{bmatrix} * \begin{bmatrix} 4 & 5 & 2 \\ 1 & 9 & 0 \\ 1 & 5 & 5 \end{bmatrix} = \begin{bmatrix} 12 & 5 & 14 \\ 8 & 36 & 0 \\ 5 & 15 & 20 \end{bmatrix} \tag{3.5}$$

In MATLAB, if you want to calculate the product of `A` and `B` element-by-element, you need to use a period sign ('.') between the first matrix and the operator:

```
>> J = [3,1,7;8,4,6;5,3,4]
>> B = [4,5,2;1,9,0;1,5,5]
>> K = J .* B
```

so that elements of K are calculated using $K(r,c) = J(r,c) * B(r,c)$. Note that there is no difference between $J * B$ and $B * J$, since the array elements are multiplied on an element-by-element basis anyway.

- ▶ Is it possible to calculate `M*N` if both `M` and `N` are arrays of size 3x2? Why or why not? If you don't know, just try it out in the Command Window!
- ▶ Same question for `M.*N`

Recall the precipitation example in Chapter 2. In that example, the precipitation had been measured for a different period in each city:

Table 3.4: Precipitation records

| **City** | **Period** | **Mean Daily Precipitation** |
|---|---|---|
| Groningen | 12 | 2.33 |
| Utrecht | 27 | 2.12 |
| Maastricht | 18 | 1.92 |

- ▶ Create two 3x1 arrays, `Period` and `MDP` (Mean Daily Precipitation), that contain the measuring period (in days) and mean daily precipitation (in mm/day), respectively for the three Dutch cities. Use matrix operations to calculate a 3x1 array `TP` with total precipitation (in mm) for every city during their respective measuring periods.

## 3.6 Concatenation

It can be useful to combine separate arrays with related data into one array. This is called concatenation.

- ▶ Type:

```
>> A = 5
>> B = 6
>> C = [A,B]
```

and check what `C` contains.

- ▶ What is the difference between typing `C = [A;B]` instead of `C = [A,B]`?
- ▶ It is also possible to add a new row and/or column to an existing array:

```
>> D = 1
>> E = 3
>> F = 4
>> G = [D,2;E,F;C]
```

- ▶ Concatenate the three arrays `Period`, `MDP`, and `TP` into one array named `PrecInfo`.
- ▶ Take 15 minutes to review what you have learned so far in chapters 2 and 3.

## Project 1. Columbia River

In this exercise you will use much more data than can be typed quickly, so you will load the data from a file. You also cannot display all results as numbers on the screen, so you will learn some of the basics of making graphical presentations.

Suppose that you have measured the stream water velocity in the Columbia River. This data is stored in a file titled 'columbia_river.mat'.

- ▶ Go to `http://blackboard.ic.uva.nl` and download the file 'pim_files.zip'. This zipfile contains all the files that you will need during this course. Unzip it to your network drive or usb-memory. It contains the folder structure displayed in Figure 3.2. As you can see, each chapter has its own folder. Some folders contain subfolders labeled according to the project number.

- ▶ Set your work directory to '\ch03_arrays\proj01_columbia'. The file 'columbia_river.mat' contains stream velocity data from the Columbia River in Oregon, averaged over total depth[1].

**TIP:** Make it a habit to clear your workspace with the `clear` command before you start a new exercise. Doing this makes your calculations faster and will help diminish the interference from unnecessary variables.

- ▶ Load 'columbia_river.mat' into the workspace (a *.mat file is a binary file, it can contain multiple variables):

[1]Data from Savini and Bodhaine (1971) US Geological Survey

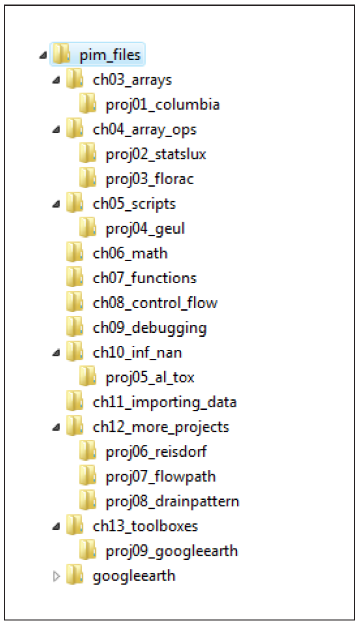


Figure 3.2: Folder structure.

```
>> load columbia_river.mat
```

▶ Check the variables you loaded for Columbia River in the workspace. Check the size and the values for these arrays in the "Workspace" window.

---

**TIP:** A calculation statement that does not end with a semicolon ( ; ) echoes the answer to the screen. This may be undesirable if you are working with large matrices, because you don't want these matrices scrolling endlessly on your screen. A statement with a semicolon carries out the same task but does not echo the result to the screen. Display the array `Sensor_1` from the Columbia River project and then again using the semicolon:

```
>> Sensor_1
```

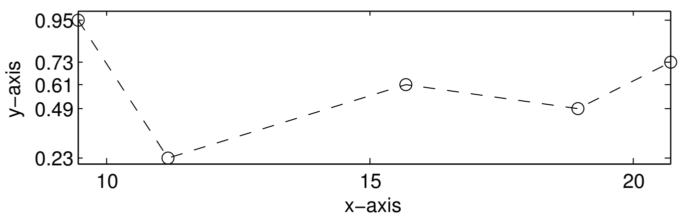
```
>> Sensor_1;
```

---

## 3.7 Simple plotting

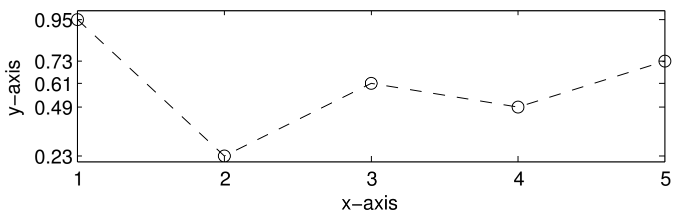
MATLAB can display all kinds of graphs. An easy-to-use visualization tool is the `plot(X,Y)` command. This command plots the elements of a vector `X` on the x-axis versus the values of a vector `Y` on the y-axis (see Figure 3.3).

If no X-vector is specified (`plot(Y)`), MATLAB visualizes the data from vector `Y` versus its indexes (see Figure 3.4 on page 17).



| X | Y |
|---|---|
| 9.46 | 0.95 |
| 11.17 | 0.23 |
| 15.69 | 0.61 |
| 18.95 | 0.49 |
| 20.71 | 0.73 |

Figure 3.3: `plot` with X and Y coordinates specified.



| X | Y |
|---|---|
| 1 | 0.95 |
| 2 | 0.23 |
| 3 | 0.61 |
| 4 | 0.49 |
| 5 | 0.73 |

Figure 3.4: `plot` with only Y coordinates specified explicitly.

► Display the data of `Sensor_1` using the plot command.

```
>> plot(Sensor_1)
```

This plot contains the time on the horizontal axis and the stream water velocity on the vertical axis. To indicate what the figure you created is about, we need to give it a title. This is done using the `title('s')` command. The variable `s` represents the text to be used in the title. Don't forget to put this text string inside single quotation marks, otherwise MATLAB will return an error.

► Give the figure a title: 'Columbia River, near Priest River Dam'. If you are not sure how to assign titles to figures, you can consult the `doc` function (see the tip below).

Just as we can put a title on a figure, it's also possible to assign text labels to the axes. This is done using the commands `xlabel` and `ylabel`.

**TIP:** There are different ways to get help in MATLAB. By far the most important and easy-to-use is the `doc` command. For instance try typing `doc mean` to read about how to use MATLAB's built-in function to calculate the mean of an array. Other commands that are sometimes helpful are: `help` and `lookfor`.

- ▶ Consult the `doc` function to find out how to assign labels to the x and y axes. Label the x-axis ‘Time (minutes)’ and the y-axis ‘Velocity (m/s)’.

- ▶ Type

```
>> legend('velocity [m/s]')
```

to create a legend for the stream velocity plot of the Columbia River.

There is a problem: previous experience with this sensor shows that a stream velocity correction is needed for this stream segment:

$$TV1 = -0.18 + 0.987 * MV1 \tag{3.6}$$

where $TV1$ is the true stream velocity and $MV1$ is the velocity as recorded by sensor 1.

- ▶ Calculate the true stream velocity of the river, measured with sensor 1.

After 66 measurements, unfortunately sensor 1 broke down. Luckily, a backup sensor, sensor 2, was available at the same stream location. The variable `Sensor_2` contains the data from this sensor. However, this one also needs a correction:

$$TV2 = -0.02 + 1.002 * MV2 \tag{3.7}$$

where $TV2$ is the true stream velocity and $MV2$ is the velocity as recorded by sensor 2.

- ▶ Correct the second data set (`Sensor_2`) with this algorithm and append the corrected sensor 2 data to the first 66 measurements gathered at sensor 1 into a new merged stream data array, called `Columbia`. Plot the result. Try to plot the 2 variables `Sensor_1` and `Sensor_2` with different colors. Consult the help documentation if necessary.

- ▶ Print the figure you created for your Columbia River stream velocities.

- ▶ Save the merged and corrected data array into a binary file by typing:

```
>> save merged_sensors.mat Columbia
```

which means: save the content of the variable Columbia in the binary file ‘merged_sensors.mat’.

---

End of Project 1.

# Chapter 4

# Array operations

MATLAB has many possibilities for array operations. In this chapter, we will look at a few of them.

## 4.1 Creating special MATLAB arrays using utility matrices

Up to this point, you created all matrices by manually filling rows and columns with values. Below are some of the MATLAB utility functions that are 'shortcuts' to special matrix formats:

Table 4.1: MATLAB utility matrices

| **Function** | **Description** |
|---|---|
| `zeros(M,N)` | Generate an array of size MxN filled with zeros |
| `ones(M,N)` | Generate an array of size MxN filled with ones |
| `rand(M,N)` | Generate an array of size MxN filled with values that are randomly chosen from a uniform distribution between 0 and 1 |
| `randn(M,N)` | Generate an array of size MxN filled with values that are randomly chosen from a normal distribution with mean 0 and standard deviation 1 |
| `eye(M)` | Generate an identity matrix of size MxM with ones on the diagonal and zeros on all other positions |

For instance, `A = [1,1,1;1,1,1]` has the same result as `A = ones(2,3)`.

▶ Clear your workspace and type

```
>> J = rand(7,5)
```

and interpret the result. If necessary use the documentation by typing:

```
>> doc rand
```

▶ To extract a submatrix from `J`, type:

```
>> Jsub1 = J(6:7,:)
```

and interpret the result.

Since in this case no column numbers are specified, **all** values of `J` that are on the 6th or 7th row of `J` are selected.

▶ Now try to extract `Jsub2`, a 7x2 array containing the last two columns of `J`.

▶ Create a 4x5 matrix filled with only 7s

▶ Create a 4x4 matrix filled with -1 but with 2 on the diagonal

▶ Create a matrix with 3 columns and 7 rows filled with uniform random numbers between 3 and -1

## 4.2 More MATLAB utility matrices

▶ Type

```
>> G = [1,2;3,4;5,6]
```

▶ Type

```
>> Gflipped = flipud(G)
```

and interpret the result.

▶ Try the following MATLAB functions as well: `fliplr` and `rot90`. Describe what happens to the array for each command. Again use `doc` to get more insight in the possibilities of these functions.

▶ Type

```
>> Gtran = transpose(G)
```

Compare the original array `G` and `Gtran`. Describe the changes.
(Note that >> `Gtran = G'` has the same effect as >> `Gtran = transpose(G)`).

▶ Create these three arrays:

- 2x3 array `[6,7,8;2,3,4]`
- 1x3 array `[1,5,9]`
- 1x3 array `[10,11,12]`

Use concatenation and manipulation commands to compose this 3x4 array from the matrices you just created:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \tag{4.1}$$

## 4.3 Statistical functions

MATLAB has a lot of built-in functions for statistical analysis of data sets. For example `min(M)` returns the minimum of a vector `M`.

- ▶ Clear the workspace
- ▶ Create a 2 x 6 array `D` using:

  ```
  >> D = [11:-3:-4;linspace(1,11,6)]
  ```

- ▶ Typing

  ```
  >> min(D)
  ```

  at the prompt will return:

```
ans =

     1     3     5     2    -1    -4
```

Interpret how these numbers were determined.

**TIP:** `ans` (short for 'answer') is a temporary variable name that MATLAB automatically generates when the user does not specify a variable name. It is overwritten every time a variable name is not specified.

Table 4.2: MATLAB Statistical functions

| **Function** | **Description** |
|---|---|
| `min(M)` | Calculates the minimum of a vector `M` |
| `max(M)` | Calculates the maximum of a vector `M` |
| `mean(M)` | Calculates the mean of a vector `M` |
| `median(M)` | Calculates the median of a vector `M` |
| `sum(M)` | Calculates the sum of a vector `M` |
| `std(M)` | Calculates the standard deviation of a vector `M` |
| `prod(M)` | Calculates the product of a vector `M` |

If `M` is an array with `R` rows and `C` columns, `sum(M)` returns a row vector. The n$^{th}$ element of this vector contains the sum of the n$^{th}$ element from all rows of `M`. In other words, `sum(M)` returns the sum of M, calculated over the first dimension, i.e. the rows. If you want to sum columns instead of rows, you can specify that the summation should be applied to the $2^{nd}$ dimension (i.e. the columns) by:

```
>> SumM = sum(M,2)
```

The second argument to the function `sum` is equal to 1 for operations on rows and 2 for operations on columns if `M` is a 2-dimensional array, but can be higher if `M` has more than 2 dimensions.

- ▶ If `X = [0,1,2;3,4,5]`, calculate `sum(X,1)` and `sum(X,2)`, as well as the sum of the entire `X` matrix, `sum(sum(X))`.
- ▶ Is there a difference between `sum(sum(X,1))` and `sum(sum(X,2))` if `X` is a 2-D array?

The other functions from the list work in a similar way. If you would like to know more about these functions, you can always try or consult `doc` or `help`.

- ▶ Calculate the sum and mean of matrix `X` over the columns.
- ▶ Calculate the sum and mean of all elements of matrix `X`.

**TIP:** If you want to calculate the standard deviation of all values of a 2-D array, you have to make it a 1-D vector first: `std(X(:))`

- ▶ Carefully read the Tip above and execute the command. Do you understand what happened?
- ▶ Why is the result of `std(std(X))` different from `std(X(:))`?

## 4.4 Naming your variables

It is not very sensible to give variables in your script the same name as any existing functions. For example, consider the following:

```
>> A = rand(3,3)
>> B = rand(10,24)
>> mean = mean(B)
```

So far, this would not generate any errors[1]. However, if you would like to calculate the mean of `A` according to:

```
>> MeanA = mean(A)
```

[1]Some newer version of MATLAB are smart enough to recognize the upcoming problem, and will raise an error at this point

MATLAB will return an error, because it is trying to make a selection from variable `mean`, which MATLAB is then trying to assign to a new variable `MeanA`. The error is generated because the selection happens to be an invalid one. This is because the values used as indices into the array `mean` are not positive integers.

**TIP:** If you want to use a particular variable name, but you are not sure if it exists already as a function, you can use `which` or `exist`. Check the documentation on these functions to see how you can use them.

**TIP:** Give your variables sensible names!

## Project 2. Statistics of the fieldwork site Luxembourg

▶ Set your work directory to '\ch04_array_ops\proj02_stats_lux'. Load the Digital Elevation Model (DEM) of the fieldwork site in Luxemburg ('dem_lux.mat') in MATLAB.

The file contains a 2-D array `DemLux` representing elevation data on a 25-meter grid. The base elevation is referenced to a 150-meter height above sea level.

▶ Have a look at the data by double-clicking the `DemLux` array in the workspace. Next, convert the data to heights above sea level by adding the base height of 150 meters. Then, using the statistical functions covered earlier, determine the minimum, maximum height, the mean and median height, and then the standard deviation of the height. Calculate the total volume of rock and soil above sea level, given that the area of a grid cell is 25x25 m.

▶ Visualize the DEM by using the MATLAB function `imagesc`. Make a clear graphical representation, including titles and a `colorbar`.

End of Project 2.

## 4.5 Relational operators

In the previous sections, you have become familiar with arrays and several operations you can perform on arrays. Also, some mathematical operators were introduced. MATLAB offers other powerful commands to analyze your arrays, such as relational operators. Relational operators perform element-by-element comparisons between arrays. The arrays must have the same size, except when comparing an array with a scalar.

▶ Clear you workspace and type

```
>> A = [8,2,5,1;7,6,9,13;11,4,12,3]
```

Suppose you want to find out what positions in the array contain values that are smaller than 8. For this small array, you could easily do that by hand, but it would be too much work for the large datasets you will normally work with in MATLAB. Imagine having to check thousands of values by hand!

▶ Type

```
>> B = (A < 8)
```

This returns a matrix the same size as `A` with ones and zeros: ones located at array positions where the value in `A` is smaller than 8, and zeros located at positions where the value is equal to or greater than 8. Arrays of this type are called 'logical' arrays[2]. Logical arrays are often used to identify positions in an array where a certain condition is satisfied. For example, in an orchard there are many trees; you could make an array `nApples` that lists the number of apples on each tree. By using logical arrays, you could give the trees that have less than 100 apples (`nApples<100`) the logical value 1 (or `true`), and those that have 100 apples or more, the logical value of 0 (or `false`).

▶ Create a matrix `C` with zeros on positions where `A` is smaller than 8, and the values of `A` on all other positions. This may not be an easy task, but with what you've learned in previous sections you should be able to derive the answer.

Of course, the `<` operator is not the only condition that you can test for. Table 4.3 on page 25 provides an overview of all conditions. As a further example,

▶ Type

```
>> G = [8,1,3,1;5,6,3,8;11,8,12,5]
```

▶ Type

```
>> H = (A == G)
```

and interpret the result.

[2]In other programming languages, this data type is sometimes known as 'boolean' (after the English mathematician George Boole; http://en.wikipedia.org/wiki/Boole)

Table 4.3: MATLAB Relational operators

| **Operator** | **Type of comparison** |
|---|---|
| `>` | greater than |
| `>=` | greater than or equal to |
| `<` | smaller than |
| `<=` | smaller than or equal to |
| `==` | equal to |
| `~=` | not equal to |

## 4.6 `find` function

The `find` function finds the non-zero elements of a matrix. If the `find` function is used in combination with a logical array, it return the positions in an array, where some condition is `true`. For the orchard example, we could find the indices of the trees that have less than 100 apples by `find(nApples<100)`. If no elements are found, `find` returns an empty matrix.

► Type

```
>> Z = A<4
>> P = find(Z)
```

returns an array `P` with indices of the non-zero values in `Z`. These indices correspond with values smaller than 4 in `A`.

```
>> [P,Q] = find(Z)
```

returns two vectors `P` and `Q`, containing the row `P` and column `Q` numbers of the non-zero values in matrix `Z`, that correspond to values of `A` that are smaller than 4.
Example

```
>> B = [4,7,2;8,9,4]
>> [P,Q] = find(B==4)
```

result:

```
P =
      1
      2
Q =
      1
      3
```

This means that the criterion `(B==4)` is satisfied at `B(1,1)` and at `B(2,3)`.

**TIP:** The `find` function returns the indices or subscripts of the values that meet the condition, not the values themselves.

▶ Use the find function to replace all 4s by 0s in the matrix `B`.

Before you go on with Project 3, review the work done through chapter 4.

## Project 3. Evapotranspiration at Florac

Water shortage has always been a problem for farmers near Florac in the Lozère province, France. To investigate water losses to the atmosphere by evapotranspiration, scientists want to analyze the effect of different vegetation types on the total evapotranspiration in this agricultural area. From the literature it is known that every vegetation type has its own transpiration rate. By combining the vegetation-specific transpiration rates with their spatial distribution, as obtained from satellite images, the contribution of every vegetation type to the total evapotranspiration can be calculated.

▶ Clear the workspace and load the file 'landuse_class.txt' located in the folder '\ch04_array_ops\proj03_florac'. This text file contains remote sensing satellite data that has been classified into three landuse classes:

Table 4.4: Satellite image classification

| **Original Value** | **Landuse** |
|---|---|
| 0 | meadows |
| 1 | evergreen forest |
| 2 | deciduous forest |

When creating graphs or plots in MATLAB, it is possible to display multiple plots in one figure. This is done by using the `subplot(a,b,c)` command, in which `a`, `b` and `c` are integer scalars, and `c` is less than or equal to the product (`a*b`). Example:

▶ Create the utility matrices `M = rand(10,10)` and `K = rand(100,100)`. Then execute the following commands one by one at the MATLAB prompt:

```
>> figure
>> subplot(1,2,1)
>> imagesc(M)
>> subplot(1,2,2)
>> imagesc(K)
```

These commands divide the figure into smaller subplots. There is one 1 row and 2 columns of such subplots. After a subplot has been activated, the variables `M` and `K` are visualized using the `imagesc` command.

▶ Why can't you use the plot command to visualize `M` or `K`?

**TIP:** In some ways, computer programming is like getting dressed: the order in which you do things can really make a difference!

- ▶ Display the variable `landuse_class` in the first subplot of a new figure using the `imagesc` and `subplot(2,3,1)` command.
- ▶ Use relational operators to create a 2-D logical array called `Meadow` with ones on positions where `landuse_class` is equal to 0.
- ▶ Plot the 2-D logical array `Meadow` in subplot (2,3,4) using the `imagesc` command.
- ▶ Create logical arrays `Evergreen` and `Deciduous` the same way you've created `Meadow`. Plot `Evergreen` in subplot (2,3,5) and `Deciduous` in subplot (2,3,6).

You will be calculating the transpiration for each landuse class for Florac for September $14^{th}$, given the transpiration rates for each landuse type:

Table 4.5: Evapotranspiration efficiencies at Florac

| **Land use type** | **Transpiration rate on the $14^{th}$ of September [$m^3/m^2$/day]** |
|---|---|
| meadows | $0.8\times10^{-3}$ |
| evergreen forest | $1.5\times10^{-3}$ |
| deciduous forest | $1.3\times10^{-3}$ |

**TIP:** You can write base-10 exponents more easily using the `e` notation. For instance, $5\times10^7$ can be written as:

```
>> 5e7
```

The grid cell dimensions are 25 meters in both the x and y directions. With the use of the logical arrays extracted earlier, it is now possible to calculate the contribution of each land use type to the total transpiration of this area.

- ▶ Create an 1x1 array containing the total transpiration for September $14^{th}$ by the deciduous forest landuse class.

**TIP:** Make it a habit to use variable names that are long enough to clarify the meaning, but preferably no longer than about 10–15 characters.

- ▶ Calculate 1x1 arrays for the transpiration of the evergreen and meadow landuse classes in the same manner you calculated the transpiration for the deciduous landuse class.
- ▶ Calculate the sum of all transpirations to get the total transpiration. If your answer is 4.1777e+004 $m^3$ then your answer is correct, otherwise try again.
- ▶ Create a vector `FTr` that contains the transpiration fraction each land use type contributes to the total transpiration.
- ▶ Use the `barh` command to display the vector `FTr` in horizontal bars in the third subplot on the first row of subplots. Use the `xlabel` command to give the x-axis a representative label. If you like, you can print the figure you created for this project.

---

End of Project 3.

Chapter 5

# MATLAB scripts

As you probably noticed during the previous exercises, entering MATLAB commands at the prompt is a difficult way to create a structured, coherent program, especially because the order in which commands are executed is important. The solution to this problem is the MATLAB script file. The script file is a user-created text file that contains a sequence of MATLAB commands that are created with the MATLAB editor.

▶ Type:

```
>> edit
```

The MATLAB text editor window will open. In this window, you can write series of command lines, edit them and save them for running in MATLAB whenever you choose to. These files are called ‘scripts’ and are saved as *.m files[1]. Whenever you type the filename of your script in the command window, the command lines in that file are executed one after the other. It is important to add comment lines (as opposed to ‘command lines’) that explain the purpose of specific parts of your program.

**TIP:** If the first character on a line is `%`, that whole line is regarded as a comment line. Comment lines are easily recognized in the MATLAB m-file editor because its font color is by default set to green. Comments are ignored by MATLAB during calculations.

An example of a MATLAB script is given in Code Snippet 5.1 on page 30. Studying existing scripts is an excellent way to learn about MATLAB programming.

[1]By default, Windows XP does not show a file's extension in Windows Explorer. However, you can change the default behavior by clicking “Start” ↝ “Settings” ↝ “Control Panel”. Next, double-click “Folder options” and choose tab “View”. In the list of items, uncheck “Hide extensions for known file types”. Open Windows Explorer to confirm that it now shows the file extension.

Code Snippet 5.1: Calculation of evapotranspiration at Florac, France

```matlab
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This programme is an example of a Matlab script for Programming in Matlab
% W. Bouten, University of Amsterdam, December 2003
% Evapotranspiration in Florac
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear         % delete the variables from the workspace
close all     % close all open MATLAB figure windows
clc           % clear the contents of the command window

% %%%%%%%% INITIALIZATION %%%%%%%%%
load landuse_class.txt

% %%%%%%%% CALCULATIONS %%%%%%%%%%%
Meadow    = landuse_class == 0;   % logical array, true for Meadow
Evergreen = landuse_class == 1;   % logical array, true for Evergreen Forest
Deciduous = landuse_class == 2;   % logical array, true for Deciduous Forest

% calculate the transpiration by counting the number of cells sum(sum(X))
% and then multiplying with the area per cell and the transpiration per unit
% area
TrDeci      = sum(sum(Deciduous))*25*25*1.3e-3;
TrEverg     = sum(sum(Evergreen))*25*25*1.5e-3;
TrMeadow    = sum(sum(Meadow))*25*25*0.8e-3;
TotTrans    = TrDeci + TrEverg + TrMeadow;

% calculate the fractions
FTrDeci     = TrDeci/TotTrans;
FTrEverg    = TrEverg/TotTrans;
FTrMeadow   = TrMeadow/TotTrans;
FTr = [FTrDeci,FTrEverg,FTrMeadow];

% %%%%%%%% OUTPUT %%%%%%%%%%%%%%%%%
subplot(2,3,1)
imagesc(landuse_class)
colorbar

subplot(2,3,4)
imagesc(Meadow)
title('Meadow')

subplot(2,3,5)
imagesc(Evergreen)
title('Evergreen')

subplot(2,3,6)
imagesc(Deciduous)
title('Deciduous')

subplot(2,3,3)
barh(FTr)
xlabel('Transpiration Fraction')
```

**TIP:** When writing scripts, it is often helpful to structure your program according to the general structure described in Code Snippet 5.2. Standardizing m-files in this way makes them easier to design, program, and interpret at a later time.

Code Snippet 5.2: General structure of script m-files

```
% HEADER describing what the m-file does, who
% wrote the program, what version of MATLAB
% it was developed on, etc.

% INITIALIZATION

clear         % clears old variables from the workspace
close all     % close all open figures
clc           % clear the command window


% DYNAMIC PART / CALCULATIONS
% this is the part of the program where most
% of the work is done



% OUTPUT & VISUALIZATION
% in this part, you can let your program write
% information to disk, save figures and so on
```

**From this moment on, you will develop your programs in the MATLAB editor.**

- ▶ Set the work directory to '\ch05_scripts'. Start a new script in which the assignments of the exercise below are executed, save it as '∗*name*∗_gargellen.m' in the work-directory (∗*name*∗ being your own last name).

- ▶ Clear the MATLAB workspace by adding the appropriate command to your m-file. After saving your file, you can execute its contents by typing the script's file name at the prompt. Note that you must omit the script's extension (.m) when typing its name in the command window. Alternatively, you might want to click the 'run. . . ' button in the editor (see Figure 5.1).

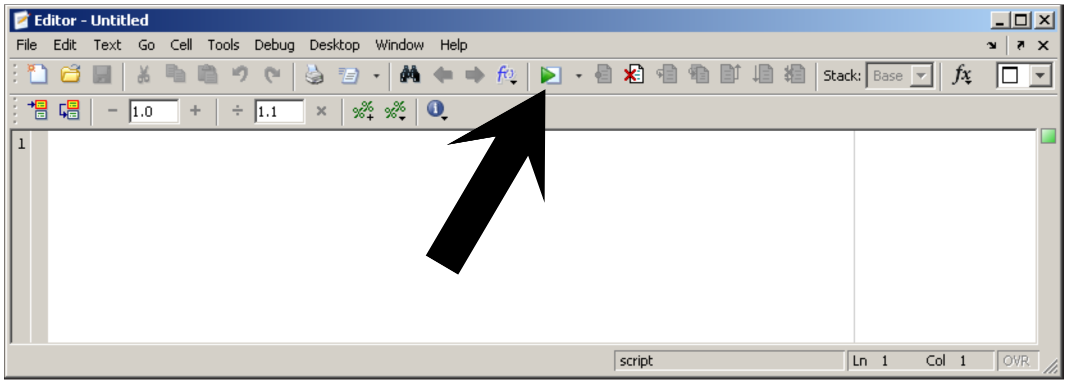


Figure 5.1: Location of the 'run' button.

The file 'demIceAge.txt' contains data on the elevation (in units of meters) of gridcells 10,000 years BP in the Gargellen valley in Austria[2]. 10,000 years ago, during the Ice Ages, the Gargellen Valley was under a glacier. After an increase in temperature, the ice melted and erosion created a major gully. The data file 'dem2001.txt' contains the elevation (in units of meters) of the Gargellen Valley as of 2001.

- ▶ Load the files 'demIceAge.txt' and 'dem2001.txt' into the workspace.

- ▶ Use the `imagesc` command to display the contents of the variables `demIceAge` and `dem2001` in subplot(2,2,1) and subplot(2,2,2). Use a `colorbar` statement to indicate the elevation.

- ▶ Give the subplots representative titles.

- ▶ Calculate the difference in height between `demIceAge` and `dem2001` to obtain a map indicating the amount of erosion (in units of meters) over the last 10,000 years. Display this difference in subplot 3, and give it a title.

- ▶ Given that the grid cell dimensions of both DEMs is 25 x 25 meters, calculate the total volume of rock that has been eroded in the past 10,000 years.

- ▶ Save and execute your MATLAB script program.

- ▶ Before proceeding to the next project, make a review of concepts covered in Chapter 5.

[2] 47.003658°N,9.955546°E

**TIP:** Good programming practices:

1. Don't forget to clear your Workspace variables when you begin to program your script files;
2. Try to limit the use of numbers in the Dynamic Part of your script (see Code Snippet 5.2 on page 31). Instead, use variables with a value that was assigned in the Initialization section;
3. Build your script by starting with the Initialization and the Output (see Code Snippet 5.2 on page 31). Then add the calculations in between;
4. Test parts of your script regularly. This is easily done by copy-pasting those parts of the script code to be tested into the "Command Window", and then executing these parts separately. Alternatively, you can run parts of your m-file by selecting them in the editor and subsequently pressing F9 (or equivalently: right-click and choose "Evaluate Selection").

## Project 4. Lead pollution in the Geul river valley

The Geul river is located in the Province of Limburg, The Netherlands. Because of past mining activities in the area around the Geul, the river water used to contain high quantities of heavy metals including cadmium (Cd), mercury (Hg), and lead (Pb). For decades, these heavy metals have been deposited in the Geul Valley. As a result, the soil in the vicinity of the Geul river has been polluted.

The Province of Limburg wants to develop the area, but can only do so after a safety assessment has been made. Specifically, they are interested in the severity of Pb pollution in part of the valley. Cd and Hg concentrations have already been measured. These two metals do not reach dangerous concentrations and will therefore not threaten the construction plans. However, the Pb concentrations may exceed the maximum allowed concentration in parts of the valley. To investigate this, the construction companies and politicians have commissioned an assessment of the severity of the Pb pollution in the Geul valley.

More than 100 soil samples have already been taken, and a map has been constructed. Now it's up to you to manipulate and analyze this data. First, let's take a look at the Geul lead pollution data. The data are available as 'geulmap.txt'.

▶ Make sure you follow the m-file standardization guidelines for this project (see Code Snippet 5.2 on page 31 and the Tip above). Set your work directory to '\ch05_scripts\proj04_geul' for this exercise.

▶ Start the m-file editor and save the empty script program as '∗*name*∗_geul.m' (with your name for ∗*name*∗) in your new work directory.

- ▶ Load the geulmap data into your MATLAB workspace. Use `imagesc` and `colorbar` commands to have a quick look at the Geul river data. Let your script visualize the `geulmap` data in the upper left subplot (see Figure 5.3).

- ▶ Initially, the plan is to clean all grid cells which have concentrations in excess of 350 mg/kg. Make a 2-D logical array `concMoreThan350` which is `true` for indices in `geulmap` that are greater than 350. Let your script visualize it in the lower right subplot.

- ▶ Let your program calculate the total cost of having to clean all cells with concentrations in excess of 350 mg/kg, when given that the area of one grid cell is 900 $m^2$, and the cost equation for one grid cell is:

$$TotalCost = 1000 + 1.7031 \times CellArea \tag{5.1}$$

  where *TotalCost* is the cost in euro, and *CellArea* is the grid cell area in $m^2$. (The correct answer is 23719578).

- ▶ Let your program round off the total cost to the nearest euro (see `doc round`).

- ▶ Just as you used concatenation to merge numeric arrays (see Chapter 3.6 on page 14), you can also let MATLAB merge character arrays. At the prompt, create the character array `string1`:

```
>> string1 = 'My age is:  '
```

  Now, create the character array:

```
>> myAge = '29'
```

  (Note the quotes). At this point, you can merge the two strings by:

```
>> stringBoth = [string1,myAge]
```

  However, you'll often find that you want to merge a character array with a numeric array, rather than two character arrays. If `myAge = 29` rather than `myAge = '29'`,

```
>> stringBoth = [string1,myAge]
```

  will not yield the proper result. We can solve this by using the `num2str` function, which is used to convert numeric values to their character array respresentation.
  For example, try out the following:

```
>> clear
>> string1 = 'My age is:  '
>> myAge = 29
>> stringBoth = [string1,num2str(myAge)]
```

- ▶ Now let your program create a title similar to the one in the lower right subplot of Figure 5.3 on page 38.

It turns out that the 350 mg/kg level was a bit too ambitious: the Province does not want to spend more than 14 million euro on cleaning the soil. It has therefore been decided that the best solution is to clean those parts where humans will be exposed to the soil (recreational areas, playgrounds and so on). After these so-called priority areas have been cleaned, the rest of the money will be used to clean as much of the valley as possible; for this part of the operation, the goal is to clean all grid cells which have a Pb concentration above a certain level.

▶ Load the priority areas from the file 'needs-cleaning.mat'. Visualize the array in the upper right subplot. Give it an appropriate title.

At this point, your program can visualize a logical array of concentrations above a certain level, and it can also display the logical map containing the priority areas. However, before we proceed with the project, let's digress for a moment to take a look at logical expressions (as you will see later, these are useful for the completion of the Geul project). After that, we'll come back to the Geul river project to finish up.

## 5.1 Logical expressions

Earlier on, we encountered relational expressions such as `A>8`, or `B~=C`. However, sometimes you want to test for more complicated relations. This is when logical expressions come in handy. Basically, there are 4 logical expressions (see also Figure 5.2 for a graphical explanation):

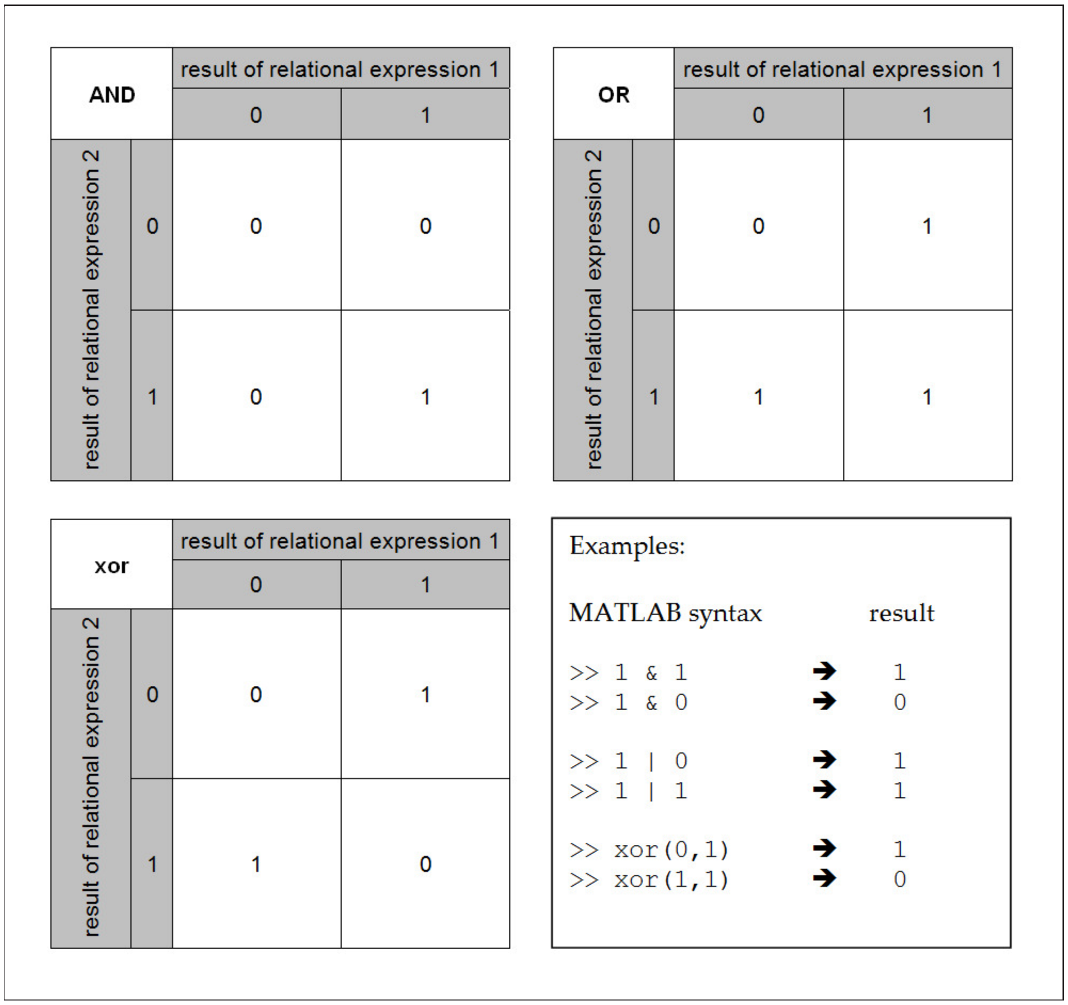
Table 5.1: Logical expressions.

| **Expression** | **Sign** | **Description** |
|---|---|---|
| AND | `A&B` | The AND expression combines entries from logical arrays `A` and `B`, returning `true` if and only if an element in `A` is `true` and the same element in `B` is also `true`. |
| OR | `A\|B` | The OR expression combines entries from logical arrays `A` and `B`, returning `true` if an element in `A` is `true` or the same element in `B` is `true` or when they are both `true`. |
| Exclusive OR | `xor(A,B)` | The Exclusive OR expression combines entries from logical arrays `A` and `B`, returning `true` if an element in `A` is `true` or the same element in `B` is `true`. It returns `false` if the element is `true` in both array `A` and `B`. |
| NOT | `~A` | The NOT expression negates the values in `A`, that is, what is `true` will become `false`, and vice versa. Is equivalent to `A==0` when `A` is logical. |

▶ Clear your workspace

▶ Type

```
>> A = [2,8,0,4,3,10];
```



| AND | | result of relational expression 1 | |
|---|---|---|---|
| | | 0 | 1 |
| result of relational expression 2 | 0 | 0 | 0 |
| | 1 | 0 | 1 |

| OR | | result of relational expression 1 | |
|---|---|---|---|
| | | 0 | 1 |
| result of relational expression 2 | 0 | 0 | 1 |
| | 1 | 1 | 1 |

| xor | | result of relational expression 1 | |
|---|---|---|---|
| | | 0 | 1 |
| result of relational expression 2 | 0 | 0 | 1 |
| | 1 | 1 | 0 |

Examples:

| MATLAB syntax | | result |
|---|---|---|
| `>> 1 & 1` | ➔ | 1 |
| `>> 1 & 0` | ➔ | 0 |
| `>> 1 \| 0` | ➔ | 1 |
| `>> 1 \| 1` | ➔ | 1 |
| `>> xor(0,1)` | ➔ | 1 |
| `>> xor(1,1)` | ➔ | 0 |

Figure 5.2: Logical expressions.

- ▶ Type

```
>> M = (A > 3) & (A < 10)
```

- ▶ Why would the OR-operator be useless in this example? What would be the result?

**TIP:** Relational and logical operations can only be performed if the arrays are of the same size.

- ▶ With this new knowledge of combining logical arrays, go back to your Geul script. What is the logical expression which allows you to create an array that has `true` on all positions that need to be cleaned? (Remember that cleaning is necessary in case of high Pb concentration, or because it's a priority area).

- ▶ If you didn't already do this, change your script in such a way that you can recalculate the total cost of cleaning by changing the value of 1 variable in your program.
- ▶ Adjust the value of this variable until it meets the budget as set by the Province of Limburg, while maximizing the cleaning objective.

## 5.2 Saving your variables to file

▶ You can let MATLAB save workspace variables to harddisk. For example, typing

```
>> save('testfile.mat')
```

will save all variables that are present in the workspace to a so-called '∗.mat-file' (extension is .mat, file type is binary) which you can load at a later time.

▶ Execute the current exercise in the command window: Create some arbitrary variables and save them to a ∗.mat file. Now issue a `clear` command. Check that your workspace is now empty. Next, load the ∗.mat file into the workspace. Check the workspace again.

Because it is usually unnecessary to save all the variables in your workspace to harddisk, you can also save specific variables only. If you want your files to be saved as text files instead of binary files, you must use the `save` option `'ascii'` to indicate the desired file format. If you choose to use the ASCII text format, your filename should have the '*.txt' extension. The `save` command can be incorporated in your script files just like any other command.

Although the `save` command can be used in a number of ways (see `doc save`), its most common forms are:

```
save('safemap.txt','SafeMap','-ascii')
```

and

```
save('valley-variables.mat','SafeMap','ValleyIO','gridCellArea')
```

The first example saves the variable `SafeMap` to a file 'safemap.txt' containing plain text (ASCII[3] essentially means 'unformatted text'. You may be familiar with files that you can read with Notepad –these are ASCII files). Remember that variable names are not allowed to start with anything but a letter character (see TIP on page 5); this way, MATLAB knows that `'-ascii'` must be an option rather than a variable. The second example saves the 3 variables to the binary file 'valley-variables.mat'. Check the MATLAB help documentation for further information on the `save` command.

▶ Let your program save the most important variables to a file on harddisk.

[3] http://en.wikipedia.org/wiki/ASCII

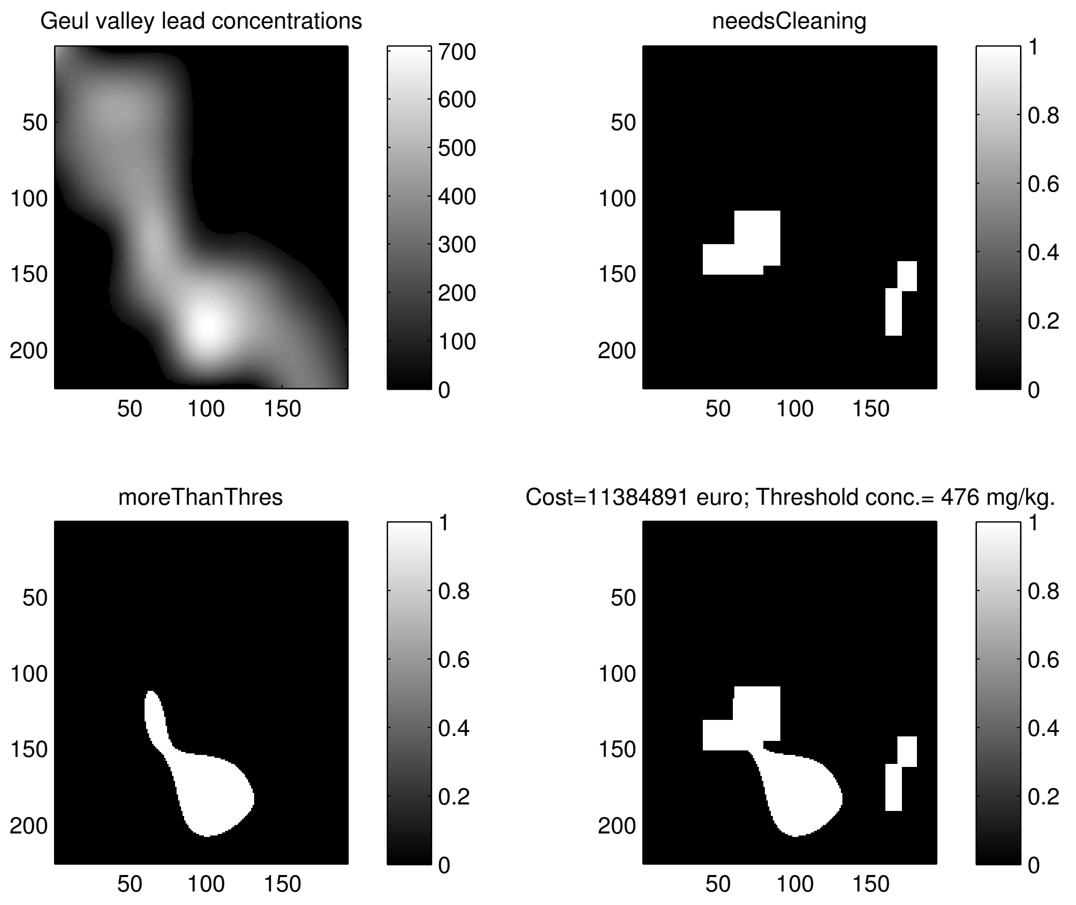


Figure 5.3: End result of the Geul pollution project. Note that the colors are different to facilitate easier interpretation when printed.

End of Project 4.

# Chapter 6

# Mathematical functions

Not only does MATLAB provide many tools and utilities to analyze arrays, it also features a lot of mathematical commands:

Table 6.1: Trigonometrical functions

| **Description** | **Example** |
|---|---|
| sine | `sin(3.1416) = -7.3464e-006` |
| cosine | `cos(3.1416) = -1.0000` |
| tangent | `tan(0.7854) = 1.0000` |
| inverse sine | `asin(1) = 1.5708` |
| inverse cosine | `acos(-1) = 3.1416` |
| inverse tangent | `atan(1.0000) = 0.7854` |

Table 6.2: Exponential functions

| **Description** | **Example** |
|---|---|
| exponential, $e^x$ | `exp(1) = 2.7183` |
| natural logarithm (base-$e$) | `log(2.7183*2.7183) = 2.0000` |
| common logarithm (base-10) | `log10(100) = 2` |
| square root | `sqrt(25) = 5` |

► Clear your workspace by manually entering the appropriate command at the prompt and type

```
>> G = [1:3:30];
```

(Answer this before executing it in the command window). Consider the following command `F = sin(G)`.

Table 6.3: Rounding functions

| Description | Example |
|---|---|
| round towards zero | `fix(-5.7323) = -5` |
| round toward $-\infty$ | `floor(-5.7323) = -6` |
| round toward $+\infty$ | `ceil(-5.7323) = -5` |
| round towards nearest integer | `round(-5.7323) = -6` |

- ▶ What are the array dimensions of `F`? Check after answering.
- ▶ Clear your workspace and define a 1 x 100 array `Trigo` that contains values ranging from -2*pi to +2*pi. It might be useful to take another look at the creation of MATLAB utility matrices covered in chapter 4.
- ▶ Calculate the second row of `Trigo`, containing the sine of the elements in `Trigo(1,:)`.
- ▶ Create a new figure and plot `Trigo`. In order to do this correctly, you must specify which vector you want to be represented on each of the axes. Assign a legend to the figure using the `legend` command `legend('sine')`. Label the x-axis ‘independent variable’ and the y-axis ‘math function’.
- ▶ Create a 1 x 60 array called `Expon`, filled with equidistant values ranging from -3 to +3.
- ▶ Calculate the second row of `Expon` filled with values that are exponents of the values in the first row ($10^x$ , in which $x$ is the element in the first row).
- ▶ Create a new figure and plot the second row of `Expon`.
- ▶ Create a new figure and use the `semilogy` command to display the second row of `Expon` in a plot with a logarithmic y-axis. Consult the help documentation for information on the use of `semilogy`.
- ▶ Type `ylabel('10^x')` and confirm that it is displayed as $10^x$, then set the color of the sine in figure 1 to `'r' (red).`

  Code Snippet 6.1 contains an example of how to use the `plot` function. (Note that it is slightly different from what you just did).

Code Snippet 6.1: Example of how to use the `plot` function

```matlab
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% % Example to show some properties of the plot function
% % W. Bouten, UvA, dec 2003
% % Course: Programming in MATLAB
% %
% % File: '\ch06_math\plot_example.m'
% % MATLAB version 2006a (win32)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% INITIALIZATION PART:
clear                                        % clear the workspace variables
close all                                    % close any open figures
clc                                          % clear the command window


% DYNAMIC PART/ CALCULATIONS:
Trigo = transpose(linspace(-2*pi,2*pi,60)); % fill first column of matrix
Trigo(:,2) = sin(Trigo(:,1));                % assign the sine to the second column
Trigo(:,3) = cos(Trigo(:,1));
Trigo(:,4) = tan(Trigo(:,1));


% VISUALIZATION PART:
figure
plot(Trigo(:,1),Trigo(:,2), 'b-')            % plot a blue (b) line (-) for the sine
axis([-8,8,-1.5,1.5])                        % set the ranges of the x- and y-axes
xlabel('independent variable')               % assign label to the x-axis
ylabel('math function')                      % assign label to the y-axis
hold on                                      % keep the plot while plotting the next
plot(Trigo(:,1),Trigo(:,3), 'k+-',...        % (the dots mean that the
     Trigo(:,1),Trigo(:,4)/10, 'r*')         % line continues)

% add a legend next to the plot, the position is
% indicated by -1, see help documentation
legend('sine','cosine','tangent/10',-1)
```

# Chapter 7

# Function m-files

MATLAB functions such as `sin`, `sqrt`, `linspace` and `find` take the variables you enter and perform the operations specified in these functions. The commands executed inside the function, as well as any intermediate variables created by these commands, are hidden. These variables exist in a part of the workspace that is separate from the top level (or 'Base' workspace) that you have worked with so far. The lower level of the workspace is sometimes referred to as 'function workspace'. In a way, function m-files are like a black box: all you see is the input that goes in and the output produced by the function. Functions provide a very powerful way to modularize your program. Modularity is useful, because it lets you structure your computer program. Because variables that are created by a function 'live' in a separate part of the workspace, they cannot interfere with variables outside the function workspace. This means that you can have a variable `X` defined in the base workspace, as well as in the function workspace. Changing the value of `X` in the function will not affect the value of `X` in the base workspace, unless you explicitly tell MATLAB that you want this by including `X` as a output variable.

Function m-files are similar to MATLAB script m-files in that they are text files with a *.m extension. Also, like script m-files, function m-files are not entered in the command window but rather are external text files created with a text editor. However, a function m-file is different from a script m-file in that it begins with the 'function definition line', which has to be the first command line in the m-file, and has to start with the word "function". This way, MATLAB knows that the commands inside the m-file need to be executed in a separate workspace.

The function definition line always has the following structure (in order of appearance):

1. the word `function`;
2. zero, one, or more output variables. If there is more than one output variable, they need to be enclosed in square brackets '`[ ]`'. Multiple variables need to be separated by commas;
3. the is-equal sign '=';
4. the name of the function. Its use here is only for completeness: When MATLAB encounters a function name in a script m-file, it starts looking for an m-file by that name as it is known by the operating system (Windows, Linux etc). For instance, when you call the function `mean`, MATLAB uses the code from a file called 'mean.m' residing

somewhere on your harddisk. Because some operating systems are case-sensitive with respect to filenames, you may only use lowercase characters in your script and function filenames.

5. the last part of the function definition line lists the input variables. This part may contain zero, one, or more input variables. The variables need to be enclosed in parentheses '`( )`'. Multiple variables need to be separated by commas.

Code Snippet 7.1 gives a few examples of valid and invalid function definition lines.

Code Snippet 7.1: Various function definitions for a file called 'calcflow.m'

```
% VALID FUNCTION DEFINTIONS FOR A FILE CALLED 'calcflow.m':
function calcflow
function CALCFLOW
function lofknwj
function calcflow(A)
function calcflow(A,B,C)
function A = calcflow(B)
function [A,B] = calcflow(C)
function [A,B] = calcflow(C,D)


% INVALID FUNCTION DEFINTIONS FOR A FILE CALLED 'calcflow.m':
function [A,B] = calc-flow(C,D)
function [A,B] = calcflow[C,D]
function [A,B] = calc flow(C,D)
```

**TIP:** Variables that are passed to a function, or that are returned by a function, are referred to as 'input arguments' and 'output arguments', respectively.

## 7.1 Writing function m-files

▶ Study the function m-file 'statsm.m' located in the folder '\ch07_functions' (see Code Snippet 7.2).

As can be seen in this example, the function definition line of a function m-file defines the m-file as a function, specifies its function name (`statsm`), and defines its input (`M`) and output (`MaxM`, `MinM`, `MeanM`) variables. Following the function definition line is a sequence of comment lines, as indicated by the percent sign. The comment lines are meant to provide information about the function m-file.

▶ Make sure you are in the right directory and type:

```
>> help statsm
```

As you can see, MATLAB returns the first block of comment lines upon typing `help`. This is useful when you forgot how a particular function works; however, it does mean that you have to provide the help comment block in the m-files that you produce. Note: any blank lines before the `%` sign in this first comment line disqualifies it from being displayed.

Ideally, the first comment block should start with some keywords (line 2 in Code Snippet 7.2), followed by a line that states how the function must be called (line 3). The help comment should also contain a description of what it does (line 4), its input and output variables, including the array type, array dimensions, and a short description (lines 5–8). Additionally, it should contain information on the author of the file (line 10), the date (line 10), and the MATLAB version that was used to develop the software (line 11).

## 7.2 Using a function in your script

In contrast to script m-files, a function m-file cannot run by itself. Instead, function m-files are usually called by a script m-file. The function's input variables must be present in the workspace before evaluating the function call. The input arguments can then be passed on to the function, where they get processed by the code in the function m-file. Code Snippet 7.3 provides an example of how the function `statsm` (see Code Snippet 7.2) may be used.

### 7.2.1 Workspaces and code re-use

▶ Open 'stats_lux.m' in the editor and run it. After the program finishes, why isn't there a variable `M` in the workspace?

Note that the arrays `MaxLux`, `MinLux`, `MeanLux` and `dem_lux` in `stats_lux` contain the same information as `MaxM`, `MinM`, `MeanM` and `M` in `statsm`. This is yet another useful feature of functions: because your function has its own workspace, it does not have to use the same variable names as your script. What's very important though, is that you have to pass your arguments in the right order for this to work correctly. The system of order-based argument passing allows you to re-use your code, without having to adapt the code inside the function.

▶ Take 5 minutes to think about what you would have to change in `statsm` if the arguments would always be called the same inside the function workspace, as compared to outside).

▶ Create your own function m-file that extends `statsm` by also calculating the standard deviation of all values in the matrix `M`. Save the altered m-file as '**name**_statsm.m'. Adapt the script in accordance with your function. Save it as '**name**_stats_lux.m'. (The correct answer for the standard deviation of the height in the study area Luxemburg is 64 m).

Code Snippet 7.2: Contents of the file '\ch07_functions\statsm.m'

```matlab
function [MaxM,MinM,MeanM] = statsm(M)
% % keywords: statistics, maximum, minimum, mean
% % [MaxM, MinM, MeanM] = statsm(M);
% % This function calculates some statistics of a 2D matrix M
% % input     M       2D numeric array
% % output    MaxM    maximum of all values in M
% %           MinM    minimum of all values in M
% %           MeanM   mean of all values in M
% % example for Programming in MATLAB
% % W.Bouten, UvA, dec 2003
% % MATLAB 6.5 Release 13
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Calculate all statistics
MaxM = max(max(M));
MinM = min(min(M));
MeanM = mean(M(:));
```

Code Snippet 7.3: Contents of the file '\ch07_functions\stats_lux.m'

```matlab
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %  % % % % %
% % This programme calculates some statistics of the study area Luxemburg
% % as a demonstration of the use of the function 'statsm'
% % W. Bouten, UvA, Dec 2003
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %  % % % % %

clear
close all
clc

% load the data
load('DemLux.txt')

% call the function and display the results in numbers and graphics
[MaxLux,MinLux,MeanLux] = statsm(DemLux)
imagesc(DemLux)
colorbar
title('Height above sea level [m]')
```

**Summary of why functions are useful**

Many script m-files contain function m-files for certain calculations. Why use function m-files if you can simply enter the calculation command lines in the script m-file itself?

Functions have some major advantages:

1. In many cases, script m-files in which all calculations are entered in one file can become too long to read and difficult to understand. Those script m-files with thousands of command lines are far from user-friendly. The use of function m-files offers the possibility to organize the program, making it easier to understand.
2. A function m-file can be called an infinite number of times within a program. Instead of writing command lines for similar calculations over and over in an m-file, one function m-file can perform these calculations every time with new input arguments and have the result returned to the main program through its output arguments.
3. Variable names in the function m-file are local and will not appear in your workspace: they exist only within the function. You have almost complete freedom in naming input and output arguments in the function itself, just follow the guidelines in the TIP on page 5. As indicated previously, use logical and easy-to-understand names for your input and output variable names.

### 7.2.2 Multiple calls

The function definition line dictates the manner in which function m-files have to be incorporated into the main program. Suppose we have a main program in which a function m-file is used to convert degrees to radians (Code Snippet 7.4).

Code Snippet 7.4: Contents of the file '\ch07_functions\slope_dir_script.m'

```
% Main program showing two applications of conv_deg2rad.

% Direction in degrees
  DirDeg = 45;

% Slope in degrees
  SlopeDeg = 12;

% Use function m-file conv_deg2rad to convert unit
% of direction to radians:
  [DirRad] = conv_deg2rad(DirDeg);

% Use function m-file conv_deg2rad to convert unit
% of slope to radians:
  [SlopeRad] = conv_deg2rad(SlopeDeg);

% End of program
```

The main program `slope_dir_script` defines the input variables to be processed by the function m-file `conv_deg2rad` (See Code Snippet 7.5). Note that `conv_deg2rad` will convert

Code Snippet 7.5: Contents of the file '\ch07_functions\conv_deg2rad.m'

```
function [OutRad] = conv_deg2rad(InDeg)
% % keywords: degrees, radians, conversion
% % [OutRad] = deg2rad(InDeg)
% % Converts input argument to radians
% % input           : InDeg       1x1 numeric array
% % output          : OutRad      1x1 numeric array
% % authors         : Spaaks & Kraal
% % date            : 09-Sep-2003
% % matlab-version  : 6.5  Release 13

% Convert degrees to radians
OutRad = (InDeg/360)*(2*pi);
```

any number to radians as long as the syntax is correct (that is, you didn't make a 'spelling' or 'grammar' mistake). For example, `conv_deg2rad` is used first to convert a direction, after which it is used a second time to convert a slope angle. Even though the meaning of `DirDeg` is different from that of `SlopeDeg`, it is not necessary to change the contents of 'conv_deg2rad.m'.

▶ Is it useful to include the `clear` command in a function m-file? Why (not)?

So far, we have encountered functions that were called by scripts. However, functions can also be called by other functions. We will see an example of this during project 6.

# Chapter 8

# Control flow

A boulder lying on a slope will slide downslope if the friction between the boulder and the hillslope decreases below a certain threshold value. Mass wasting of this type can occur when surface friction is lowered by rainfall:

```
if friction < threshold
    % call function rockfall
else
    % no rockfall
end
```

This is an example of how you can let MATLAB decide which parts of a program should be evaluated: depending on the outcome of the test `friction < threshold`, either the `if...else` bit or the `else...end` bit is evaluated. Besides the dynamic decision-making capability of the `if-else-end` structure, MATLAB provides code structures which allow for the repeated execution of a set of commands (a so-called 'loop'). Termination of the loop can be because it is set to run a predetermined number of times, or because a certain condition is satisfied. The former is known as a `for`-loop, while the latter is known as a `while`-loop. `if-else-end` structures, `for`-loops and `while`-loops are examples of 'control flow' structures. Because these constructions often encompass multiple MATLAB commands, they are mostly used in scripts and functions (as opposed to at the prompt).

## 8.1 for-loops

`for`-loops allow a group of commands to be repeated a fixed, predetermined number of times. The general form of a for-loop is:

```
for x = rowvector
   % assignment statements which usually
   % employs variable x in some way
end
```

▶ Execute the following (in a new script):

```
clear
close all
clc

for k=1:5
    A(k) = k^2
end
```

Be aware that the above is equal to the so-called 'vectorized' way of programming:

```
A = [1:5].^2
```

It is up to you to decide which form is better suited for specific cases. Be aware that MATLAB performs its calculations faster if you use vectorization. Loops, however, are usually easier to understand. Also, some problems can not be adequately handled by vectorized programming.

### 8.1.1 Malthusian population growth

One of the first researchers into population dynamics was Thomas Malthus[1], (1766–1834). In his 'Essay on the Principle of Population' (1798) Malthus observed that in nature plants and animals produce far more offspring than can survive, and that Man too is capable of overproducing if left unchecked. This exponential growth is known as Malthusian population growth; the rate at which a population grows is directly proportional to its size.

Pierre-François Verhulst[2] (1804–1849) was a Belgian mathematician who generalized the Malthusian model by allowing for the fact that populations encounter internal competition as they grow within a closed environment, and this competition has a tendency to retard the rate of growth. His idea says that while the population will continue to grow as time goes on, the rate at which it does this growing gets smaller. This is a slightly more realistic approach than that of Malthus, whose idea actually predicts that populations will grow exponentially, and without bound –a prospect that defies physical limitations.

The script m-file '\pim_files\ch08_control_flow\malthus.m' contains part of the program that calculates population size over time based on the ideas of Malthus and Verhulst. The exponential growth and competitive growth decline are expressed in Equations 8.1–8.2:

$$V = 1 - \frac{P_{now}}{P_{max}} \tag{8.1}$$

$$P_{next} = P_{now} + r * P_{now} * V \tag{8.2}$$

where $V$ is the Verhulst term, $P_{max}$ is the maximum sustainable population size, $P_{now}$ is the current population size, $P_{next}$ is the population size for the next time step, and $r$ is the growth factor.

▶ Complete the Verhulst term in 'malthus.m'.

▶ Study 'malthus.m' and identify which role each variable has. Next, implement Equation 8.2 in the for-loop.

▶ In Section 8.1, we saw that some problems can be vectorized. Is this such a problem? Why (not)?

[1]http://en.wikipedia.org/wiki/Thomas_Malthus
[2]http://en.wikipedia.org/wiki/Pierre_Fran%C3%A7ois_Verhulst

## 8.2 while-loops

As opposed to a `for`-loop that evaluates a group of commands a fixed number of times, a `while`-loop evaluates a group of statements as long as a certain condition is true. The general form of a `while`-loop is:

```
while expression
   % assignment statements
end
```

The assignment statements between the `while` and `end` statements are executed as long as the `expression` is true. This implies that the `while`-loop can only be terminated if one of the variables in `expression` is changed *inside* the `while`-loop.

▶ Study the example script m-file in Code Snippet 8.1:

Code Snippet 8.1: Example of a `while`-loop

```
% Example while-loop
Steps = 0;
Value = 0;
MaxValue = 200;

while Value < MaxValue
    Value = Value + rand;
    Steps = Steps + 1;
end

disp(['Maximum value reached. Steps = ',num2str(Steps)])
```

Note that the variable `Steps` needs to be converted from numerical value (double array) to text (char array) by the `num2str(Steps)` command to allow for concatenation with the character array `'Maximum value reached.  Steps = '`. After that, the resulting character array can be output to the command window by the function `disp` (see MATLAB documentation).

**TIP:** By clicking in the command window and simultaneously pressing Ctrl-c, you can stop calculation anytime.

▶ Write a script m-file with a `while`-loop that never terminates.

▶ How would you characterize the difference between a `for`-loop and a `while`-loop? When would you use one or the other? Give an example of each.

**TIP:** One of the most common mistakes concerning control flow is the never ending `while`-loop. A helpful rule of thumb is that the variables which are in the `expression` have to be changed inside the `while`-loop, otherwise it can never terminate.

## 8.3 if-else-end

Many times, sequences of commands must be conditionally evaluated based on a relational test. Recall our first control flow example:

```
if friction < threshold
    % call function rockfall
else
    % no rockfall
end
```

Based on the outcome of the relational test `friction < threshold`, either the command sequence between the `if` and `else` statements is called, or the command sequence between the `else` and `end` statements is called.

When there are three or more alternatives, the `if-else-end` construction takes the form:

```
if expression1
     % evaluated if expression1 is True
elseif expression2
     % evaluated if expression2 is True and
     % expression 1 is false
elseif expression3
     % evaluated if expression3 is True and
     % expression 1 and 2 are false
elseif expression4
     % evaluated if expression4 is True and
     % expression 1, 2 and 3 are false
else
     % evaluated if none of the expressions is True
end
```

In this construction, only the commands associated with the first true expression encountered are evaluated; all following relational expressions are not tested, and the rest of the `if-else-end` construction is skipped. Furthermore, the final `else` command may or may not appear.

An application of `if-else-end` will appear in the next chapter.

# Chapter 9

# Using the debugger

In the process of programming m-files, it is inevitable that errors, also known as bugs, will occur. MATLAB provides some functionality to assist in *debugging* m-files. Two types of errors can occur in MATLAB expressions: syntax errors and run-time errors.

## 9.1 Syntax errors

Syntax errors (such as misspelled variable names, misspelled function names, missing quotes or parentheses, incorrect use of special characters, etc.) are found when MATLAB is preparing to run the program. MATLAB flags these errors immediately upon execution of the program and provides feedback about the type of error encountered and an estimate of the line number in the m-file where it occurs. Given this feedback and some experience, these errors are usually easy to spot.

## 9.2 Run-time errors

Run-time errors (programming errors such as zero divided by zero) are generally more difficult to find, because you can't tell just from looking at the code that it will generate an error. For example, your script may contain an addition `A = B + C`, which you would not expect to contain an error. However, if the arrays `B` and `C` do not have the same size, MATLAB will raise an error.

## 9.3 Manual debugging

In script m-files, you can monitor all calculation steps and variables by executing the command lines of the script m-file one by one in the command window. In this way, errors can usually be spotted quickly. In function m-files however, variables only exist as local variables living in their own part of the workspace away from the base workspace. This can make monitoring of the calculations and variables within functions very difficult.

There are several approaches to debugging function m-files. For simple problems, the easiest way to find bugs is to use a combination of the following:

1. Remove semicolons within the function so that intermediate results are displayed in the command window;

2. Add statements that display variables of interest within the function;
3. Change the function m-file into a script m-file by placing a % before the function definition statement at the beginning of the function m-file. When executing as a script file, the workspace is the MATLAB workspace, and thus it can be interrogated after the error occurs. Declare input variables in the Command Window, otherwise they won't be present in the workspace;
4. Inserting a `pause` or `keyboard` command (e.g. in loops) enables you to view changes in your variables step by step (Refer to the documentation for the use of these functions).

When an m-file consists of multiple functions, or functions are called within functions, these methods are often not enough to efficiently debug your programs. For such problems, you must use the MATLAB Debugger.

## 9.4 Debugging in the Editor

► Use the editor to open '000_debugscript.m' and '000_pfilt.m' (located in the folder '\ch09_debugging'). Click on "File" in the editor's menu and select "Save as. . . ".

► Substitute your own last name for the zeros in the filenames.

► Adapt the function call of `000_pfilt` in `000_debugscript` in accordance with the new name of the function m-file.

`debugscript` is a simple program that uses the function `pfilt` to filter a data set based on the deviation from the mean. However, the program is not functioning properly. Unfortunately, it is always very difficult to trace the bugs in someone else's program. You will need the MATLAB debugger to help you.

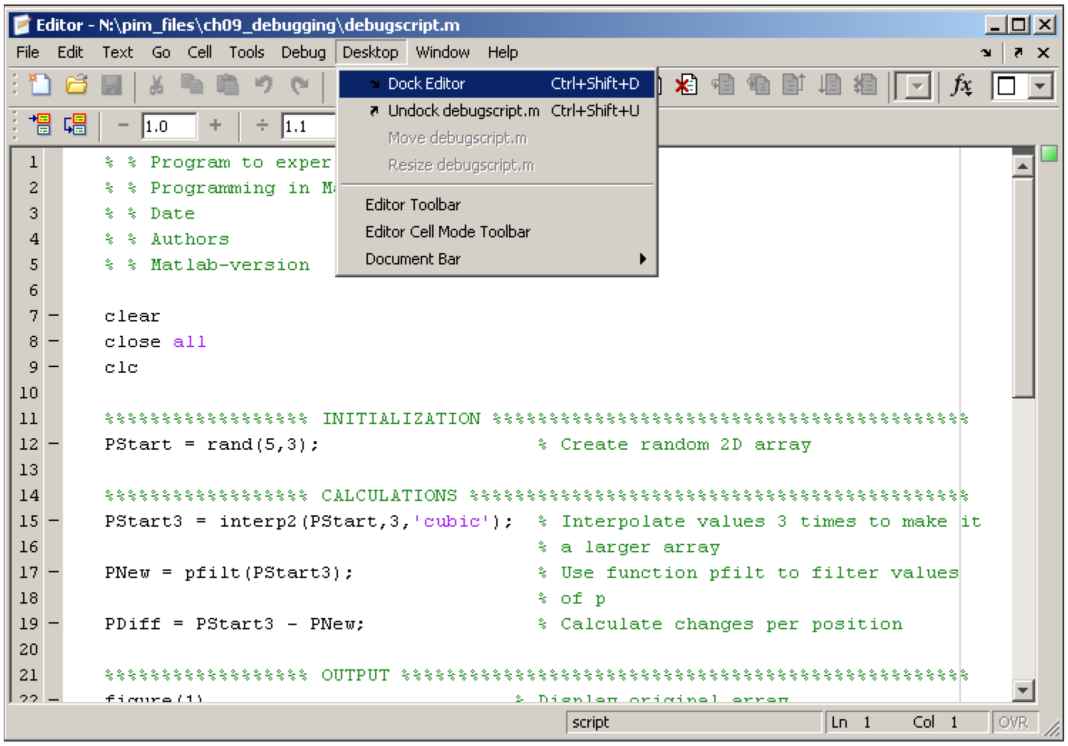
► Check for errors in the script `debugscript` by running it from within the m-file editor.

► To be able to use the Debugger effectively, you need to arrange the GUI in such a way that you can see the editor window as well as the command window and the workspace. An easy way to do so is by clicking the "Dock Editor"-menu item under "Desktop" in the m-file editor's menu (see Figure 9.1).

Two errors occur somewhere in `pfilt`. Because the variables in `pfilt` live in the function workspace, monitoring of these variables is not straightforward. You can use the debug options in the Editor to trace and fix these errors.

Note: to be able to use the debugging functionality, make sure that you have started the m-file editor from the command window using the `edit` command. Also, if the editor window has been docked into the MATLAB window, make sure that it is the active window by clicking anywhere in your m-file.

In order to use the debugger, you have to set breakpoints first. These breakpoints define where calculations will be (temporarily) stopped in the debug mode. You can set a breakpoint by clicking on one of the dash signs in the m-file editor's margin. Alternatively, you can move your cursor to a line and press the F12 button to set a breakpoint on that line. If you want, you can set multiple breakpoints. After a breakpoint has been set, a marker appears in the margin on that line (see Figure 9.2). The color of this marker can be red or gray. If it is red, that means that the script in which your breakpoint has been set, was not

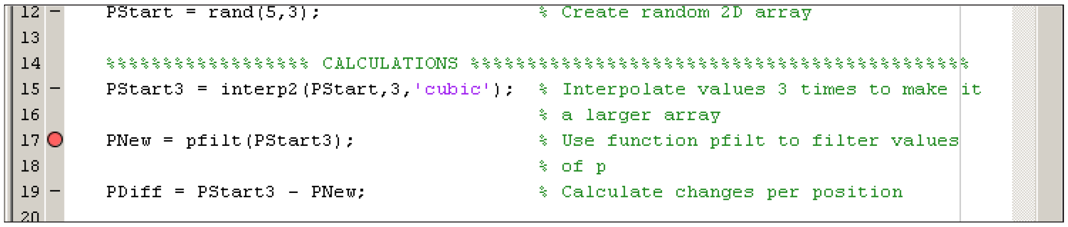

Editor - N:\pim_files\ch09_debugging\debugscript.m
File Edit Text Go Cell Tools Debug Desktop Window Help
Dock Editor Ctrl+Shift+D
Undock debugscript.m Ctrl+Shift+U
Move debugscript.m
Resize debugscript.m
Editor Toolbar
Editor Cell Mode Toolbar
Document Bar
1.0
1.1
script
Ln 1
Col 1
OVR

```
% % Program to exper
% % Programming in M
% % Date
% % Authors
% % Matlab-version

clear
close all
clc

%%%%%%%%%%%%%%%%%% INITIALIZATION %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
PStart = rand(5,3);                    % Create random 2D array

%%%%%%%%%%%%%%%%%% CALCULATIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
PStart3 = interp2(PStart,3,'cubic');   % Interpolate values 3 times to make it
                                       % a larger array
PNew = pfilt(PStart3);                 % Use function pfilt to filter values
                                       % of p
PDiff = PStart3 - PNew;                % Calculate changes per position

%%%%%%%%%%%%%%%%%% OUTPUT %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
figure(1)                            % Display original array
```

Figure 9.1: Docking the m-file editor.

changed since you saved it. If it is gray, that means that you have changed your m-file since the last time you saved it.

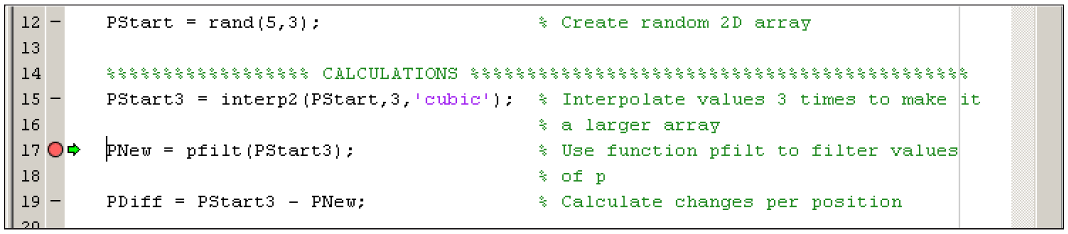
**TIP:** Before running your file, you should always make sure that you saved it.



```
PStart = rand(5,3);                    % Create random 2D array

%%%%%%%%%%%%%%%%%% CALCULATIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
PStart3 = interp2(PStart,3,'cubic');   % Interpolate values 3 times to make it
                                       % a larger array
PNew = pfilt(PStart3);                 % Use function pfilt to filter values
                                       % of p
PDiff = PStart3 - PNew;                % Calculate changes per position
```

Figure 9.2: A breakpoint has been set.

By pressing F5 or clicking the "Run" button, the commands in the active m-file will be evaluated line by line until the end of your m-file is reached, or until the first breakpoint, whichever comes first. Breakpoints are usually set a few lines up from where you suspect an error. If you don't have any idea as to where an error occurs, you can set your breakpoint on the first line of the file. If you press F5 when any breakpoints exist in your m-file or in one of the functions it uses, MATLAB will go to the debug mode. In the command window, the >> is changed to `K>>` to indicate the debug mode (`K` for 'keyboard'). If the commands in your m-file can be evaluated successfully up to the point where the breakpoint has been set, a green arrow appears on the breakpoint's line (see Figure 9.3). After the breakpoint is reached, control is returned to the user and variables can be checked and commands entered at the prompt.



```
12 -     PStart = rand(5,3);                        % Create random 2D array
13
14       %%%%%%%%%%%%%%%%%% CALCULATIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
15 -     PStart3 = interp2(PStart,3,'cubic');   % Interpolate values 3 times to make it
16                                                  % a larger array
17       PNew = pfilt(PStart3);                     % Use function pfilt to filter values
18                                                  % of p
19 -     PDiff = PStart3 - PNew;                    % Calculate changes per position
```

Figure 9.3: `debugscript` has been evaluated successfully up to the breakpoint.

Up to this point, debugging has been very similar to simply entering your commands one by one at the prompt. However, as soon as one of those commands happens to be a function (in which an error occurs), you will not be able to monitor the variables in the function's workspace by simply entering them at the prompt, since a function is somewhat like a black box: only the function's output arguments are returned to the base workspace. Because you cannot see the changes in the function's variables, finding the error in your function can become very difficult. The debugger, however, offers some useful tools to assess where the error is coming from.

If a breakpoint has been set and the commands can be evaluated successfully up to the breakpoint (in other words: if you have a green arrow), you can use "Step" from the "Debug" menu to go through your m-files line by line. Alternatively, you can press the F10 button.

▶ Set a breakpoint at the very first command line (i.e. next to `clear`) and run through your file line-by-line with the step function. Monitor how new variables are added to your workspace while you step through it.

**TIP:** You might have to right-click your workspace window and select "Refresh" to see the changes to your variables.

If you try to step past the function call line at line 17, you will notice that MATLAB returns an error. Apparently, the error is generated by some command used in the `pfilt` function. Besides stepping through your m-file line-by-line by pressing F10, you can also step into functions with the input arguments that were calculated in the script.

▶ Set a breakpoint at the function call line of `pfilt` and run your script m-file up to that point. List the variables that are present in your workspace.

▶ Now, select "Step in" from the "Debug" menu (or, equivalently, press F11) to step into the `pfilt` function with the input argument(s) necessary for the successful execution of `pfilt`.

► The green arrow should now be right next to the first command line in `pfilt`. Compare your workspace with the list of variables you made a moment ago. Where did all the variables go? And where did `In01` come from? What information does it contain?

The great advantage of using "Step in" is that the workspace will contain the locally declared variables, which enables the explicit, step-by-step monitoring of your variables, even within function m-files.

---

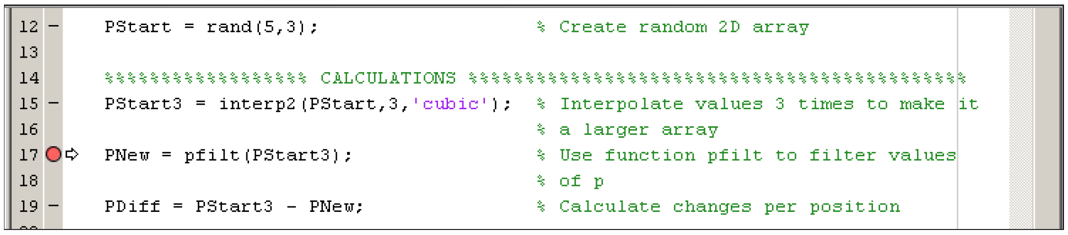
**TIP:** Pressing 'F5' or clicking the "Run" button when the active m-file is a function, will return an error:

```
???  Input argument 'In01' is undefined.
```

This is because trying to run a function on its own will cause the function's input arguments (in this case `In01`) to remain undefined.

---

If you step into a function, the green arrow in the parent m-file (in this case `debugscript`) changes into a white arrow (see Figure 9.4). This indicates that `debugscript` is not the file that you are currently in. Therefore, any variables displayed in the workspace window do not belong to this (parent) m-file.



```
PStart = rand(5,3);                    % Create random 2D array

%%%%%%%%%%%%%%%%%% CALCULATIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
PStart3 = interp2(PStart,3,'cubic');   % Interpolate values 3 times to make it
                                       % a larger array
PNew = pfilt(PStart3);                 % Use function pfilt to filter values
                                       % of p
PDiff = PStart3 - PNew;                % Calculate changes per position
```

Figure 9.4: A white arrow in the parent m-file shows that it is not the active file.

► Use the "Step" function in `pfilt` and check your workspace. Let the debugger execute command lines in `pfilt` one at a time and continue to monitor the variables in your workspace. Monitoring of the variables in the workspace involves:

1. variable values;
2. variable names (pay attention to lowercase/uppercase)
3. variable dimensions (check the workspace column 'size') or type `>> whos`
4. variable class (what is the data type of the variable under consideration?) For an overview of data classes, check Chapter 3.

► What problem arises when `MinCrit` and `MaxCrit` are calculated? What is the reason for this problem? Alter the command lines to solve this problem.

You have now corrected one of the errors in `pfilt`. However, when running `debugscript`, MATLAB still returns an error message. Considering the output arguments of the function `pfilt`, what command would you expect to be on the last line of this m-file?

► Correct the last error in `pfilt`. Use the debug tools if necessary.

"Exit Debug Mode" from the "Debug"-menu will exit the debug session and invoke the normal mode indicated by the >> sign. The debug mode can be exited at any time during debugging.

**TIP:** The "Stop if errors/warnings..." in the "Debug" menu allows you to stop calculations right before an error or warning occurs. This functionality can be especially helpful when debugging loops.

Before proceeding with the next chapter, make a review of the concepts covered in Chapters 6–9.

# Chapter 10

# Infinity and missing values

Since a computer is a discrete and finite machine, certain operations are undefined:

▶ Type

```
>> 10/0
```

The result of this operation is `Inf`. `Inf` is the arithmetic representation for positive infinity (`-Inf` for negative infinity). Infinity results from operations like division by zero and overflow (calculations that lead to results too large to represent as conventional floating-point values).

▶ Type

```
>> 0.0/0.0
```

The result of this (illegal) operation is `NaN`: Not-a-Number. A `NaN` result is obtained because of mathematically undefined operations. Undefined operations that produce `NaN`:

1. any arithmetic operation on an array containing `NaN`
2. division, such as `0/0` and `Inf/Inf`

Often `NaN`s occur in imported data.

▶ Type

```
>> A = [1,2,3;4,NaN,6;7,8,9]
```

▶ Try to calculate the mean and standard deviation of `A`.

Because arithmetic operations on `Inf` or `NaN` produce `Inf` or `NaN`, it is necessary to remove or replace these values in arrays before further calculations.

▶ Type

```
>> B = [123,Inf,43.2,18,28,NaN,pi,exp(1),-Inf,4]
```

▶ Type

```
>> Bnan = isnan(B)
```

and

```
>> Binf = isinf(B)
```

and interpret the results. Take a look at the data classes of the variables listed in the workspace.

▶ Use `isnan`, `isinf` and `find` in combination with logical and relational operators to create an array `BFinite` that contains only those values of `B` that are not `NaN` or `Inf`. Use only one command line. The result `BFinite` should be an 1x7 array.

## Project 5. Aluminum toxicity

Often, the data you import or create during programming contain errors. Values such as `NaN`, `-Inf`, and `Inf` can occur in a data set as a result of measurement errors or illegal mathematical operations. Because statistical operations are impossible on these values, it is sometimes necessary to remove them from your data before further calculation.

Aluminum (Al) is a common element found in sandy soils. At pHs up to about 4.5, $Al^{3+}$ is the most common form of Al. At higher pHs, the Al can form complexes and become immobile in the soil environment. The 'free' $Al^{3+}$-ion is mobile and can be transported easily with the soil water. This way plants can take up Aluminum, which is an important nutrient. However, high $Al^{3+}$ concentrations can become toxic to plants, inhibiting growth or even causing death. Evaluating concentrations of $Al^{3+}$ is important in determining the toxicity hazards in ecosystems. In the Dutch dunes, many experiments have taken place that monitor $Al^{3+}$ concentrations in the soil water. We have obtained the data of one of these experiments, which we will use for the following exercises.

▶ Set your work directory to '\ch10_inf_nan\proj05_al_tox' and start a new script called '∗*name*∗_al_tox' (in which ∗*name*∗ is your last name). Let the script load and visualize the data from the file 'al_conc.txt'. This file contains data on daily concentration of $Al^{3+}$ in soil water for a period of 365 days. Call the new variable `AlConc` using: `AlConc=load('al_conc.txt')`

As you can see, the data contains outliers (extreme values due to measuring errors or extreme conditions), `NaN`s and zero values where measurement errors have occurred.

To get a more accurate picture of the trend of the $Al^{3+}$ concentration over time, the data must be filtered. In this project, you will:

1. replace `NaN`s and zero values with the mean of surrounding values;
2. calculate the moving average (MA) of the $Al^{3+}$ concentration over time.

The MA is calculated as the mean of the value itself and a predetermined number of neighboring elements to the left and right of this value. For instance, the 3-term MA at time step 24 is calculated based on the mean of the values of time step 23, 24, and 25. The data is thus smoothed, making it easier to spot trends in the data. For this project, we will calculate the 9-term MA for the `AlConc` array. At the end, the original data, the data without outliers and the smoothed data are all displayed.

Create a flow diagram of how you think you can tackle the problem. If you are confident that you can solve the problem, feel free to use your flow diagram and ignore the steps explained below (You can always come back here for a step by step approach if your plan didn't work out).

▶ First, alter your script so that the `AlConc` data is visualized in the first subplot of a figure with three subplots beneath each other.

▶ Create two logical arrays: the first which is true at the locations where the concentration is zero, the second is true where there are `NaN`s. Assign your variables sensible names.

▶ Combine these two arrays with a logical expression and then use the `find` function to identify the indices of the locations that need to be interpolated.

Instead of replacing these zeros and NaNs with a predetermined value, we have to calculate the mean of the 2 cells surrounding the corrupt value. Because this has to be done for multiple elements, the use of a loop is required.

Write the loop that calculates the new value for all elements of an array `AlConcFilled` that you must create. The easiest way to do so is in these steps:

1. Copy the contents of `AlConc` to your new variable `AlConcFilled`.
2. Create the statement that calculates the new value to replace the `0` or `NaN` at a given index, based on the two neighboring cells.
3. Make the statement more generic by expressing the previous step in terms of a variable rather than numbers.
4. Use a `for`-loop to iterate over the indexes that need to be changed.

If you like, you can also avoid a `for`-loop altogether and perform your calculations using a vectorized way of programming (see page 49).

▶ Is this method valid for any distribution of zeros and `NaN`s within an array? Why or why not? If you think not, give an example where the method would not be able to calculate a valid mean value to replace a zero or `NaN`.

▶ Can you think of a way to solve the shortcoming? Describe the principle in words or in command lines.

For calculating the moving average, we can use a similar method as for the filling process we used to make `AlConcFilled`. The 9-term MA is calculated based on the 4 array elements to the left, 4 array elements to the right, as well as the middle element. For the next exercise, you need to copy and paste the loop you used to calculate `AlConcFilled`, and adapt it to calculate a new array `AlConcMA9`, which contains the 9-term moving average of Aluminum concentrations.

▶ For `AlConcFilled`, we iterated over the holes in the data, but for `AlConcMA9`, this works differently. Adapt the `rowvector` bit (see page 48).

▶ Inside the loop, adapt your command to calculate the average of the 4 array elements to the left, 4 array elements to the right, and the middle element.

▶ Display the final `AlConcMA9` in subplot 3 in the figure. Add a title and label the axes.

- Does `AlConcMA9` structurally over- or underestimate the $Al^{3+}$ concentration? How did you check this?
- The threshold between toxic and non-toxic $Al^{3+}$ concentrations is 70 $\mu$mol/L. If the `AlConcMA9` data is used, will this underestimate or overestimate the period in a year in which toxic $Al^{3+}$ levels are reached?

  So, even though the MA offers more insight in the trend in a certain data series, it can underestimate hazards because extreme values are filtered.
- Alter the program in such a way, that the number of terms for calculation of the MA can be set dynamically. The program user has to be able to change the calculation to a 5, 11, or 15-term MA by changing one value in the initialization part of your program (see page 31).
- In addition to the MA line in subplot 3, use magenta dots to visualize the data from `AlConcFilled`. Refer to Project "Columbia river" on page 15 if you forgot how to visualize multiple data sets in one figure.

  If your script works correctly, it should yield a picture like that of Figure 10.1.

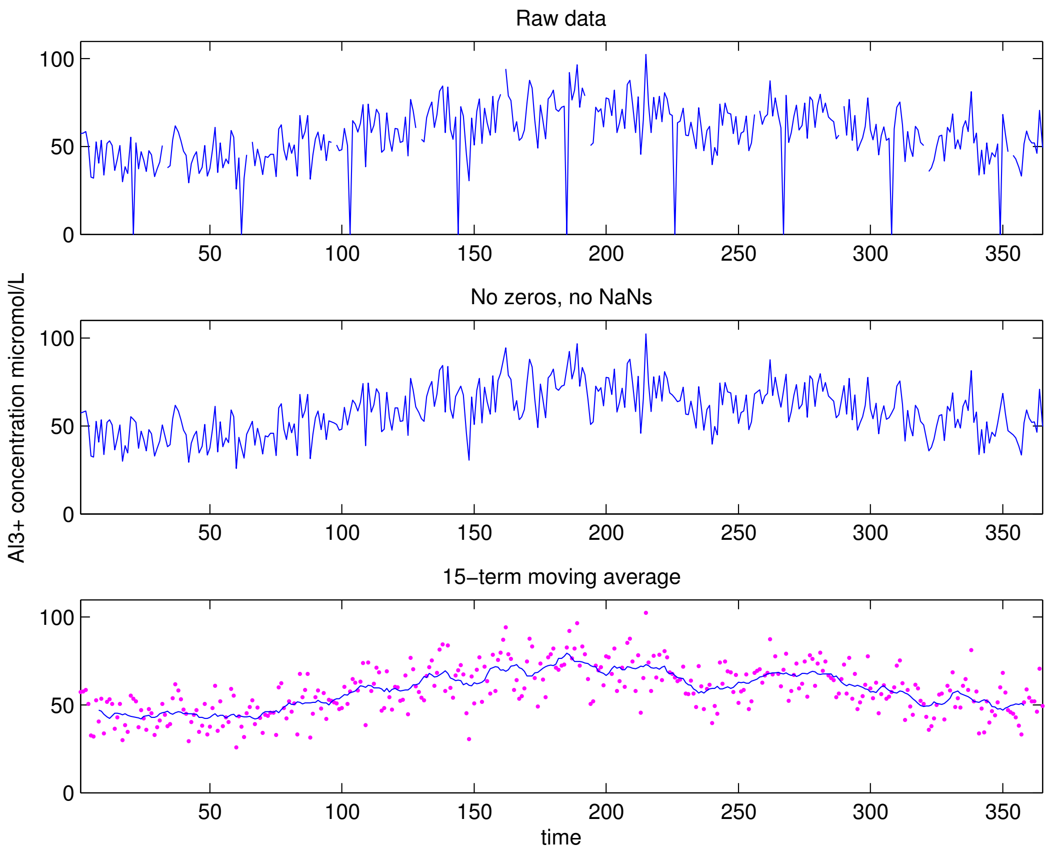


Figure 10.1: End result of Project 5.

# Chapter 11

# Importing data

The data files you have used until now consisted mainly of numeric data. However, data is usually a mixture of both text and numeric content. Because of this, it is often not possible to use the `load` command to load these files into the MATLAB workspace. However, as long as the mixed data has a fixed format, MATLAB can make data available for calculation using the function `textread`.

▶ Set your work directory to '\ch11_importing_data' and take a look at the contents of the 'soildata.txt' file.

```
1  testfile for textread
2  J.H. Spaaks & P.Kraal
3
4  soil profile       : A234.3
5  date               : 15-Sep-2003
6
7  soil horizon       : nominal class - FAO classification
8  depth              : depth of top of soil horizon
9  thickness          : thickness of soil horizon
10 root density class : 1=very high,2=high,3=intermediate,
11                      4=low,5=very low,6=no roots
12
13 soil horizon    depth    thickness    root density class
14 [-]             [m]      [m]          [m/m3]
15 Ah, 0,  0.15,  1
16 Bs, -0.15,    0.18,  3
17 Bt, -0.33,  0.09,  5
18 1C, -0.42,       2.07, 5
```

At first glance, this file doesn't seem to have a fixed format. Yet it does: the first 11 lines contain the description of the data, line 12 is empty, line 13 contains the variable names, line 14 the units, and lines 15 to 18 consist of the data itself with a comma to separate fields. This formatting enables us to load the file into MATLAB with `textread`. In its simplest form, `textread` expects the following syntax:

```
[A,B,C] = textread('filename','format')
```

where `A`, `B` and `C` are the variables in which the data from the fields of `'filename'` will be stored, and `'format'` is a character array (see Table 3.1 on page 9) in which you specify how you want MATLAB to interpret the data from `'filename'`. The format string is composed of smaller parts, each of which must start with the `%` sign. Each of the parts has the same structure: after the `%` sign, there can be an optional number, followed by a letter indicating how you want MATLAB to interpret the information that you are reading (see Table 11.1). At this point, the usage of `textread` is probably still rather vague, so let's

Table 11.1: Format string values.

| Format string | Description |
|---|---|
| `d` | Reads an integer value |
| `f` | Reads a floating point value |
| `c` | Reads a character value |

take a look at an example. The correct syntax for extracting the data from 'soildata.txt' is listed in Code Snippet 11.1. This way, you can tell MATLAB from which file it needs to

Code Snippet 11.1: Example of `textread`. Note that `'...'` breaks the command line for easier reading and printing.

```
[Hor,D,Th,RtDns] = textread('soildata.txt','%2c%f%f%d',...
                              'headerlines',14,...
                                'delimiter',',')
```

read (`'soildata.txt'`), and which format it should use (`'%2c%f%f%d'`). With this format string, you subsequently read 2 characters, a floating-point, another floating-point, and an integer, before MATLAB goes on with the next line. Because there are 4 `%` signs, there must also be 4 variables: `[Hor,D,Th,RtDns]`. If at this point we would try to read the data with:

```
[Hor,D,Th,RtDns] = textread('soildata.txt','%2c%f%f%d')
```

MATLAB would raise an error, because it will start to read at the very first line of the file unless you specify where to start. This is why the parameter `'headerlines'` was included in Code Snippet 11.1: it tells MATLAB how many lines need to be skipped before the format string can be applied. For the case of 'soildata.txt', there are 14 headerlines that need to be skipped, hence the `14` directly after `'headerlines'`. Note that we are no longer using `textread` with:

```
[A,B,C] = textread('filename','format')
```

but instead we are using the form with which you can pass options, or as they are formally known 'parameter/value pairs':

```
[A,B,C] = textread('filename','format',parameter1,value1,...
                                        parameter2,value2)
```

In the same way as we are assigning the value 14 to the parameter `'headerlines'`, we can specify the column delimiter symbol by means of the `'delimiter'` parameter. For the case of 'soildata.txt', the column delimiter symbol is set to the comma sign `','`. Besides `'headerlines'` and `'delimiter'`, `textread` has a lot more functionality to offer. For further information, including some examples, you can consult the `doc` function.

▶ Make sure you understand the `textread` example above, especially the way in which the format string is used. Now try to insert an asterisk somewhere in the format string, to ignore one of the fields from 'soildata.txt' (Refer to the documentation on how to use this asterisk). Remember that the number of fields you are trying to read must be equal to the number of output arguments, otherwise MATLAB will raise the error below:

```
??? Number of outputs must match the number of unskipped input fields.
```

- ▶ Now that you have practiced a little, it's time to try something more ambitious. Open 'knmi_dh.txt'[1]. This file contains information on 14 meteorological parameters monitored from 01-Jan-2001 to 12-Aug-2003 in Den Helder, The Netherlands (station number 235; see excerpt on page 65). Use `textread` to load the prevailing wind direction (DDVEC) into the MATLAB workspace as `WDirDeg`. For the moment, let you command ignore all columns other than DDVEC.
- ▶ Use the function `conv_deg2rad` (see page 46) to create a vector `WDirRad` with `DDVEC` in radians. In order to use the function, copy it to your work directory.
- ▶ Plot `WDirRad` using the `rose` function. Consult the documentation for details. Add a title.
- ▶ Make an additional figure in which you visualize rose diagrams for January and July using subplots. Limit yourself to the years 2001 and 2002. Use logical arrays and `find` to make your selections. You may want to alter your `textread` format string to also read the dates that a particular measurement was taken.
- ▶ Choose three variables from the KNMI data set that you find interesting. Write a program that reads these data from the KNMI file and visualizes them in the way most suitable for the type of data displayed.

[1] Data from www.knmi.nl

```
DEZE GEGEVENS MOGEN VRIJ WORDEN GEBRUIKT MITS DE VOLGENDE BRONVERMELDING WORDT GEGEVEN:
KONINKLIJK NEDERLANDS METEOROLOGISCH INSTITUUT (KNMI)

THESE DATA CAN BE USED FREELY PROVIDED THAT THE FOLLOWING SOURCE IS ACKNOWLEDGED:
ROYAL NETHERLANDS METEOROLOGICAL INSTITUTE

STN = stationsnummer / WMO-number = 06... (235=De Kooy,260=De Bilt,280=Eelde,290=Twenthe,310=Vlissingen,380=Maastricht)
YYYYMMDD = datum / date (YYYY=year MM=month DD=day)
DDVEC = overheersende windrichting / prevailing wind direction in degrees (360=North, 180=South, 270=West, 0=calm/variable)
FG  = etmaalgemiddelde windsnelheid / daily mean windspeed in 0.1 m/s (let op! inhomogene reeks door meethoogte wijzigingen /
      inhomogeneous series due to measuring height changes; zie / see: www.knmi.nl/samenw/hydra)
FHX = hoogste uurgemiddelde windsnelheid / maximum hourly mean windspeed in 0.1 m/s
FX  = hoogste windstoot / maximum wind gust in 0.1 m/s
TG  = etmaalgemiddelde temperatuur / daily mean temperature in 0.1 degrees Celsius
TN  = minimum temperatuur / minimum temperature in 0.1 degrees Celsius
TX  = maximum temperatuur / maximum temperature in 0.1 degrees Celsius
SQ  = zonneschijnduur / sunshine duration in 0.1 hour (-1 for <0.05 hour)
SP  = percentage van de langst mogelijke zonneschijnduur / percentage of maximum possible sunshine duration
DR  = duur van de neerslag / precipitation duration in 0.1 hour
RH  = etmaalsom van de neerslag / daily precipitation amount in 0.1 mm (-1 for <0,05 mm)
PG  = etmaalgemiddelde luchtdruk / daily mean surface air pressure in 0,1 hPa
VVN = minimum opgetreden zicht / minimum visibility (0=less than 100m, 1=100-200m, 2=200-300m,..., 49=4900-500m, 50=5-6km, 56=6-7k
NG  = bedekkingsgraad van de bovenlucht / cloud cover in octants (9=sky invisible)


  STN,YYYYMMDD,DDVEC,   FG,  FHX,   FX,   TG,   TN,   TX,   SQ,   SP,   DR,   RH,   PG,  VVN,   NG

  235,20010101,  177,   88,  110,  170,   40,    9,   76,    0,    0,   71,   88, 9944,   22,    8
  235,20010102,  180,   81,  130,  200,   77,   42,  101,   12,   16,   29,   19, 9895,   58,    7
  235,20010103,  217,   74,  110,  180,   63,   42,   86,   51,   66,   25,   17, 9988,   56,    6
  235,20010104,  185,   86,  130,  190,   72,   50,   82,    0,    0,   73,  105, 9925,   19,    8
  235,20010105,  201,   75,  110,  190,   69,   62,   96,    0,    0,   76,  123, 9854,   24,    8
  235,20010106,  241,   67,   90,  140,   64,   55,   72,   61,   78,    5,    1, 9962,   58,    5
  235,20010107,  223,   70,  100,  140,   61,   50,   75,   34,   43,    7,    4,10058,   45,    5
  235,20010108,  251,   58,   80,  110,   60,   49,   68,   15,   19,   26,   28,10096,   40,    6
  235,20010109,  258,   41,   60,  100,   56,   32,   74,   24,   30,   33,   21,10131,   20,    6
  235,20010110,   74,   50,  100,  130,   23,    2,   36,    0,    0,    0,    0,10148,    1,    7
  235,20010111,   68,   62,   90,  130,   22,   14,   41,   68,   86,    0,    0,10251,   50,    2
  235,20010112,   67,   26,   50,   70,   30,  -13,   58,   21,   26,    0,    0,10296,    5,    4
  235,20010113,   75,   38,   50,   70,   -4,  -23,   11,   57,   71,    0,    0,10367,    2,    4
  235,20010114,   80,   50,   80,  120,    9,  -36,   41,   18,   22,    0,    0,10362,    2,    6
  235,20010115,  104,   66,   80,  110,  -11,  -20,   -1,   69,   85,    0,    0,10288,   45,    3
  235,20010116,  104,   43,   50,   70,  -18,  -26,   -8,   70,   86,    0,    0,10251,   37,    1
  235,20010117,   98,   30,   40,   60,  -32,  -43,  -21,   55,   67,    0,    0,10204,   22,    3
  235,20010118,   89,   25,   40,   50,  -22,  -39,   -7,   12,   15,    0,   -1,10197,    6,    6
```

# Chapter 12

# More projects

## Project 6. Reisdorf river dam

To obtain a reliable source of drinking water and energy, the river Sûre was dammed near Esch-sur-Sûre during the 1960s. The resulting Lake of the Upper Sûre is now one of the most popular vacation spots in Luxembourg. The successful exploitation of this artificial lake has persuaded the city council of Reisdorf to construct a dam of their own. They expect that the fresh water, energy and flourishing tourist industry will bring them health and wealth. The city council has asked a team of experts from the University of Amsterdam to research the consequences of the erection of such a dam. In this exercise, we will simulate the flooding of the Ernz Blanche river valley.

- Set your work directory to the folder '\ch12_more_projects\proj06_reisdorf'.
- Load the files into the workspace using:

```
% load the DEM of the Ernz Blanche valley:
EBD = load('ebd.txt');

% load the file with the location and dimensions of the dam:
DamHeight = load('dam_height.txt');

% load the monthly supply of water:
EBSupply = load('ebsupply.txt');
```

- Visualize the Ernz Blanche DEM.
- The file 'dam_height.txt' contains data on the dimensions and location of the dam. Visualize the dam by using `imagesc`, then use the zoom-in feature to inspect the top rows where the Ernz Blanche leaves the map. The correct surface elevation data -after construction of the dam- is obtained by adding the values in `DamHeight` to `EBD`.

To analyze the consequences of the dam, you need to have insight in the water input into the system. Based on historical data, we have a dataset ('ebsupply.txt') with the volumes of water [m$^3$] that will flow into the lake each month once the dam is closed.

▶ Visualize the data from 'ebsupply.txt'.

The first 3 years after construction, the dam is kept closed to allow the water level to rise behind the dam for a profitable dam discharge.

▶ Calculate the volume of water that has accumulated behind the dam in these 3 years.

We would like to know which area is flooded after these three years.

▶ Why is it not possible to calculate the water level directly from the water volume?

In contrast to the difficulty in calculating the water level from the water volume, the reverse problem is quite easy: suppose you would know the water level `h`, the grid cell dimensions and the grid cell elevation (the DEM), it would be relatively easy to calculate the water volume beneath this water level. In this exercise, we will exploit this to approximate the true water level. A function has already been created that calculates water volume under a given water height, but accidentally it was messed up. The command lines are intact, but the order is wrong. The code can be found in the file 'mess.txt'.

▶ Put the lines in 'mess.txt' in the right order and save the file as '∗*name*∗_calcvol.m' (with your last name for ∗*name*∗). Refer to Chapter 7 if you forgot how a function is structured.

**TIP:** Variables that occur after the '='-sign must have been assigned previously; new variables can only occur before the '='-sign.

▶ To test if you changed the order of the lines correctly, type

```
>> sum(sum(calcvol(241.1318,EBD+DamHeight,25,25)))
```

(don't forget to add your last name to the function call). This should return 1.0000e+007.

▶ If `h`, `CellSizeX` and `CellSizeY` are in meters, what are the units of this result?

To be able to assess which grid cells of the DEM are flooded by the water in the lake after three years, you need to calculate the water level (above sea level) given the calculated water volume. Since it is not possible to calculate `h` directly, you must use an approximation method. This approximation method is used to try out a number of assumed water levels. For each assumed water level, you can calculate the water volume that is associated with it using `calcvol`. The water volume that is associated with the assumed water level is then compared to the actual water volume of the lake (which we can get from the values in `EBSupply`). If the assumed volume is too low, we need to adjust the assumed water level upward; if the assumed water volume is too high, we need to adjust the assumed water level downward. With each adjustment, the interval containing the true water level decreases. We need to keep adjusting until the interval has become sufficiently small. An efficient way for adjusting the value of water level is the so-called 'bisection method'. The first step of the bisection method is to define the initial lower `hLo` and upper `hUp` bounds on the water level. At any point during the approximation, the interval [`hLo`,`hUp`] is as small as possible, given the knowledge that has been collected so far about the true water level `hAct`.

▶ Determine the lowest (bottom of the lake) and highest (maximum height in the DEM) water levels. These values form the initial estimates of `hLo` and `hUp` (*Step 1*, see Figure 12.1).

▶ `hMid` is defined as the average of `hLo` and `hUp`. Calculate the corresponding water volume with `calcvol` (*Step 2*, see Figure 12.1).

The actual water volume is either between the volume associated with `hLo` and `hMid`, or between `hMid` and `hUp`, meaning that the actual water level is also in that interval. Since you now know whether the actual water level `hAct` is in the lower part or the upper part, you eliminate the part that does not contain `hAct` by redefining either `hLo` or `hUp` (*Step 3*, see Figure 12.1).
In Step 3 of the figure, `hLo` remains unchanged with respect to its position in Step 2, whereas the former `hMid` (see Step 2) is now defined as `hUp`, since you found out that the actual `hAct` is certainly not in the Step 2-interval [`hMid`,`hUp`]. Steps 2 and 3 are repeated until the interval [`hLo`,`hUp`] has become small enough to meet your preset criterion.

▶ Finish Figure 12.1 by drawing lines in the lower right subplot. Assign each line a label `hLo`, `hMid`, and `hUp`.

In the next few exercises, you will implement the bisection technique in a function that can be used to approximate the actual water level associated with a given water volume behind the dam. This function is subsequently used to approximate the water depth of the lake for each month during the first three years after the dam was erected. The function that you will be developing uses the function `calcvol` that you already made.

▶ First think of the input variables that you need and the output that you want. Together this determines the function call.

▶ A rough setup of the program is given in the function m-file 'outline.m'. Save this file as '∗*name*∗_approxh.m' (where ∗*name*∗ is your last name) and edit this file until it produces the right result as shown in Figure 12.2.

We are not only interested in the inundated area after three years, but we would like to know the inundated area for each month after the closure of the dam and as long as the water level has not reached the upper level of the dam.

▶ Start a new script called '∗*name*∗_reisdorf.m' (with your last name for ∗*name*∗). Let your script visualize the water depth in the valley for each month after the closure of the dam, starting from the dry situation and continuing until the valley is filled up to the upper level of the dam. Use the `approxh` function and the `calcvol` function that you have already written.

So far, we have used `imagesc` in its simplest form: `imagesc(X)`. This form visualizes the array `X` using a gradient of colors. The colors are automatically set in such a way that they range from the minimum of `X` to the maximum of `X`. However, sometimes you want to set these color limits yourself. For this, you can use `imagesc` with a second input argument: `imagesc(X,C)`, where `C` is a 1x2 numeric variable, with the user-specified color limits. For instance, in the Reisdorf example of Figure 12.2, the minimum color has been set to 0, whereas the maximum color has been set to 45. The command to do this is: `imagesc(WaterDepth2D,[0,45])`.

▶ Use the 2-argument form of `imagesc` to set the color limits on your water depth figure.

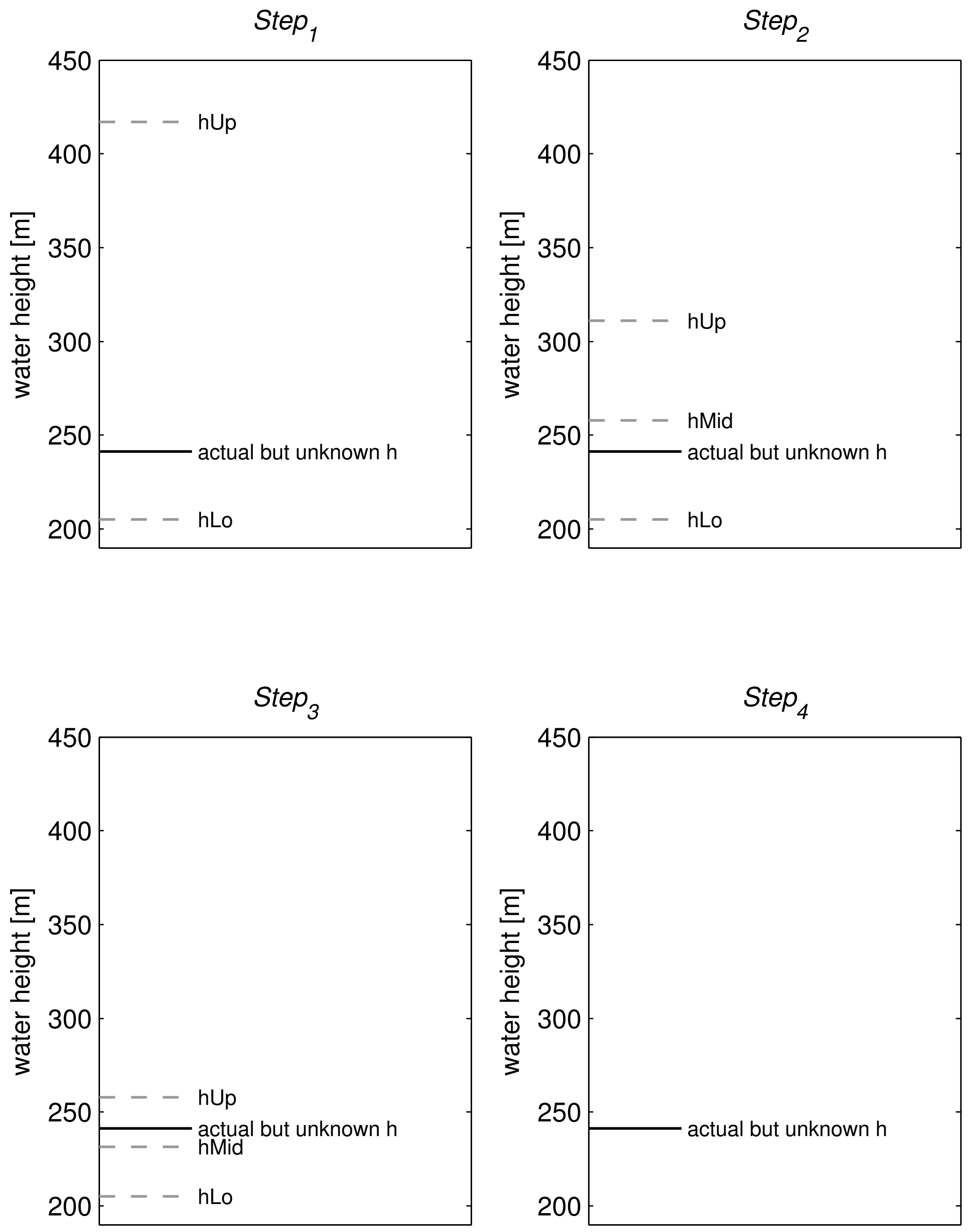


Figure 12.1: Graphical demonstration of the bisection method.

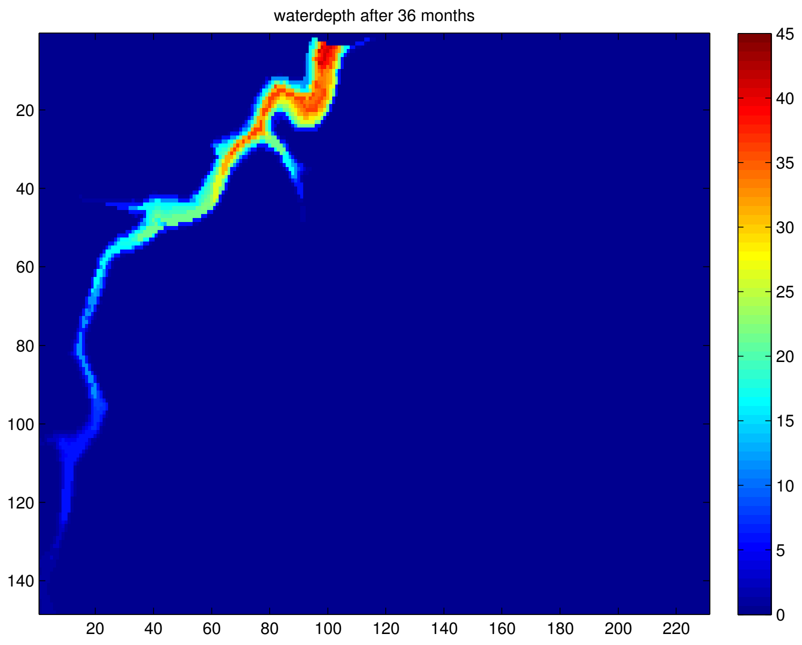


Figure 12.2: Distribution of water depths 36 months after the dam was closed.

## 12.1 Making movies

Extensive calculation results often ask for some kind of animated output. For example, the filling of the reservoir could be visualized as a function of time. In this section you learn how you can save these animations in movies that can be shown without having to run the MATLAB program again.

A movie consists of frames, just like a traditional celluloid movie. You need to initialize the movie by including the following line in the initialization part:

```
aviObj = avifile('testmovie.avi','fps',5,...
                 'compression','none');
```

This example initializes a movie file 'testmovie.avi'. Using the `'fps'` (for 'frames per second') option, you can specify at what speed you want to play the movie later. In the example, a frame rate of 5 frames per second is used. Furthermore, you can specify the compression algorithm that MATLAB will use. For now, we will set `'compression'` to `'none'`, but we will return to this later in Section 12.1.2. Running the above command will create a variable `aviObj` of data class 'avifile object'. (You can check this in the workspace). See `doc avifile` for more details.

After the avifile object has been initialized, you can add as many frames as you like (as long as there is available space on your harddisk). After issuing the visualization commands (such as `imagesc`, `scatter`, `surf` or `plot`), you need to capture the frame and add it to your avifile object variable. A generic structure of a script that creates a movie is included in Code Snippet 12.1.

Code Snippet 12.1: Generic structure of a script that creates an *.avi movie.

```matlab
clear
close all
clc

% INITIALIZATION
figure
aviObj = avifile('testfile.avi','fps',5,'compression','none');

% DYNAMIC PART (WITH VISUALIZATION)
for k=1:100 % some iteration

    % some calculations that affect variable X go here

    % visualize the new X:
    imagesc(X)

    % force MATLAB to finish drawing the figure before continuing:
    drawnow

    % get the active part of the figure:
    frame = getframe(gcf);

    % add this frame to aviObj:
    aviObj = addframe(aviObj,frame);
end

% close the avi file object:
aviObj = close(aviObj);
```

### 12.1.1 On the need for closure

It is crucial that your program gets to closing the avifile object. If you forget to include this part, or if the program crashes before it gets to closing the avifile, you end up with an *.avi file (in this case 'testmovie.avi') which cannot be played. This is because media players will wait for MATLAB to release its handle on the file (which it never will, since the program crashed). If this is the case, you need to issue a `clear all` *at the prompt*. Note that you will generally not include the `clear all` in your script, since it makes debugging impossible. For this course, we will *only* use `clear all` when dealing with movies that have not been finalized. In all other situations, you need to use `clear`.

▶ Change your program in such a way that it creates a movie of the filling of the artificial lake for each month for the first 3 years.

▶ Re-read the part about `for`-loops and `while`-loops on page 48–50. Now change your script in such a way that it keeps calculating the monthly water depth map for as long as the water level is lower than the level of the dam.

### 12.1.2 Compression

As you may have noticed, making a movie without using any compression can generate big files, especially when you are capturing many frames. This is where compression algorithms come in handy. Compression algorithms, or codecs[1], are used to decrease file size, while maintaining as much of the original quality as possible. In this respect, they are similar to what MP3 is for audio, what JPEG is for photos, or what ZIP is for data. Codecs are well worth looking into, because a good codec can decrease a file by a factor 100 without compromising the movie quality!

▶ In your work directory, there is a codec file 'mpg4c32.dll', which can be used to compress your movie file. In order for this to work, you need to set the `'compression'` option to `'mpg4'`. Make the necessary changes to have your program use this compression algorithm.

---

End of Project 6.

## Project 7. Flow path

The characteristics of a drainage network play a great part in determining how water moves through a basin and consequently affects issues such as water quality and quantity (including flooding). Drainage patterns dramatically affect people and the environment downstream regardless of administrative borders, often over long distances, and on a very large scale with long-lasting implications.

In this project, you will analyze water movement in a drainage basin. You will develop a program that calculates the pathway of water for a given DEM, from an arbitrarily chosen starting point to the lowest point of the landscape.

▶ Set your working folder to '\ch12_more_projects\proj07_flowpath', start a new script called '∗*name*∗_flowpath.m' (with your last name for ∗*name*∗ ) and visualize 'valleydem.txt'.

▶ If you would drop a ball in `valleydem(10,4)`, where would it end up?

To determine water movement in this area, you can use the principle of 'steepest descent': this principle is based on the assumption that water movement between cells is driven by the maximum gradient between the origin cell and its 8 surrounding cells.

In order to calculate this maximum gradient, you need to know the difference in height between the origin cell and the surrounding cells (`DeltaH`) and the distance from the surrounding cells to the origin cell (`Dist`). These distances are not all equal.

▶ Given that the cell dimensions are 25 x 25m, fill the array below with the distances.

[1] http://en.wikipedia.org/wiki/Codec

$$\texttt{Dist} = \begin{bmatrix} \dots & \dots & \dots \\ \dots & 0 & \dots \\ \dots & \dots & \dots \end{bmatrix} \tag{12.1}$$

Do you think that you could now write a script to calculate the path water follows from some given starting point to the lowest point? If so, try it. Start with making a design first and then write your program. If you don't think you can do it just yet, follow the guidelines below.

- ▶ What would be the statement to select a 3x3 submatrix `H` from `valleydem` if the position of the center cell in `H` corresponds with `valleydem(68,55)`? If your command is correct, `H` should be equal to:

```
H =
    113.8968   111.6601   109.4644
    115.4084   113.1661   110.9636
    116.9417   114.6987   112.4948
```

- ▶ Create a 3x3 array `DeltaH` with the height differences between the cells and the center cell.

- ▶ Create a 3x3 array `Grad` containing the gradients between the origin cell and its neighbors. The gradient is difference in height divided by distance.

- ▶ Set the value of the center cell in `Grad` to 0 to avoid calculation complications with NaN. If you do not want the warning message in your screen, include the following line in your program, just before the line that causes the warning:

```
warning('off','MATLAB:DivideByZero')
```

Remember that divide-by-zero warnings serve a purpose, so make sure you turn them back on after the line that raises the warning. You can do this by:

```
warning('on','MATLAB:DivideByZero')
```

- ▶ If one of the gradients in `Grad` is negative, what does that mean?

Now you have a 3x3 array with the values of the gradients of the surrounding 8 cells. The steepest descent-theory states that water flows (only) in the direction of the cell that has the lowest (most negative) gradient. So, the next step is to determine which of the surrounding cells will be the one into which the water flows.

- ▶ Create a 3x3 logical array `LocMin` with value 1 on the position where `Grad` has its minimum value.

- ▶ Determine the subscripts of the minimum value within `Grad`. Call them `r` and `c`, respectively.

Keep in mind that the location `[r,c]` is the position of the minimum value in `Grad`, not in `valleydem` (see also Figure 12.3).

- ▶ What is the range of valid values for `r`? And for `c`?

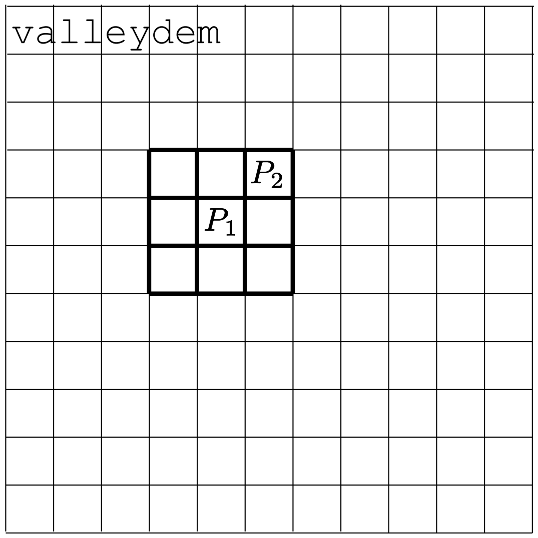


Figure 12.3: The position of `H`, `Dist`, `Grad` within `valleydem`. $P_1$ is the center cell of `H`, `Dist`, and `Grad`, with subscripts r=2 and c=2. The same position in `valleydem` is indicated by (`CurrentRow,CurrentCol`). Position $P_2$ is the position for which the gradient is most negative, i.e., the direction of steepest descent.

- ▶ If $P_1$ is the position (`68,55`) in `valleydem`, what is the position of $P_2$?
- ▶ If $P_1$ is the position (`CurrentRow`,`CurrentCol`) in `valleydem`, what is the position of $P_2$?
- ▶ We can use the subscripts of $P_2$ (with respect to `Grad`) to determine its position within `valleydem`. Alter your script in such a way that `CurrentRow` and `CurrentCol` are updated based on the value of `r` and `c`.

  At this point, we have all the necessary constituents for repeating the following steps in a for-loop:

  1. From `valleydem`, select the 3x3 subset around the current location;
  2. Calculate the location of lowest gradient;
  3. Make the location of lowest gradient the current location.

- ▶ Implement the above mentioned steps in your script m-file. Refer to the program outline given in Code Snippet 12.2 for guidance on how to structure your program. The result of your program should look like Figure 12.4. Use concatenation commands to keep a record of the value of `CurrentRow` and `CurrentCol`.

- ▶ Is it possible to start calculations at position (1,1) of `valleydem`? Why (not)?

  At this point, the script m-file calculates the new row and column for 25 iteration steps (see Code Snippet 12.2). However, it is unrealistic that water will stop flowing after 25

Code Snippet 12.2: Design of a script that calculates the flow path of water through a catchment

```
clear        % clear old variables from the workspace
close all    % close all open figures, if there are any
clc          % clear the command window

% % % % % % % % % INITIALIZATION % % % % % % % % % % % % % %
% initialization of necessary variables
%
% load the DEM
% determine the size of the DEM
% define start position CurRow and CurCol and initialize the track
% calculate the matrix Dist

% % % % % % % % % DYNAMIC PART % % % % % % % % % % % % % % %
% create a for-loop for 25 iterations
% select the submatrix H
% calculate DeltaH
% calculate Grad
% determine the row and column of the lowest Grad
% determine the next position in the DEM and add this to the track
% end the loop

% % % % % % % % % VISUALIZATION % % % % % % % % % % % % % %
% visualize ValleyDem
hold on % (for keeping the DEM on the background, see >> doc hold)
% visualize track on top of ValleyDem
```

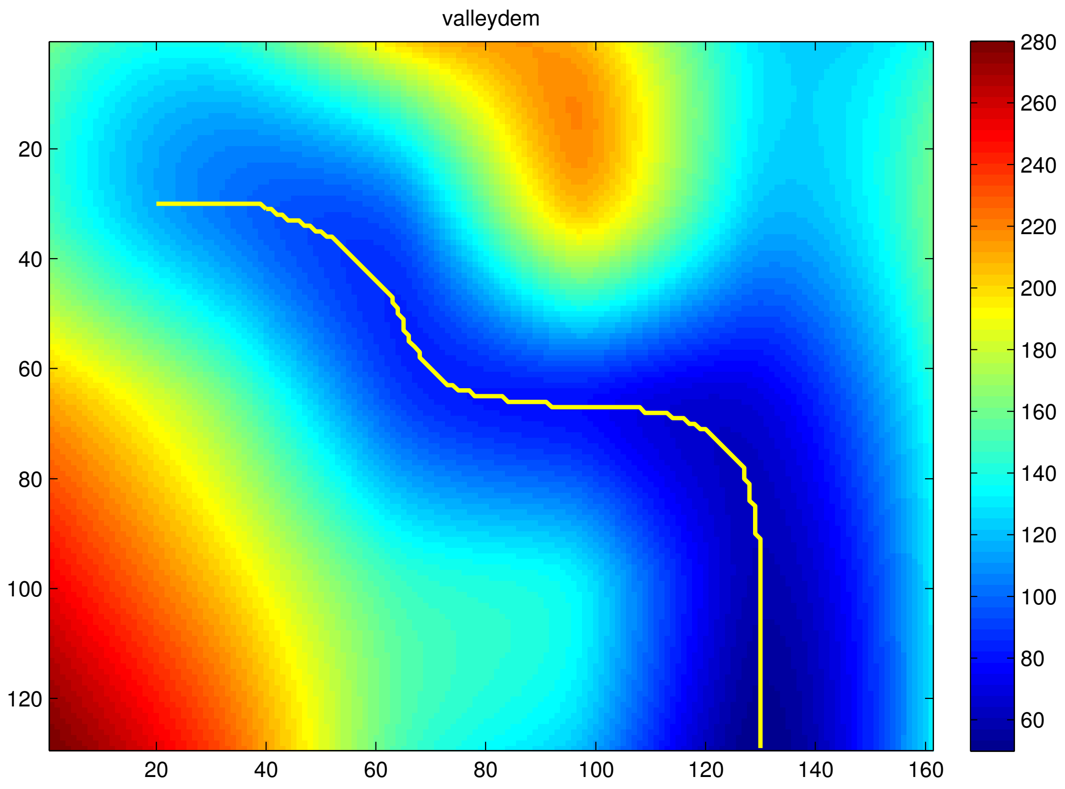


Figure 12.4: Flow path derived from the DEM using the steepest-descent method. Starting point is `valleydem(30,20)`.

steps. What you actually want, is continuous calculation until the borders of the system are reached.

As opposed to the `for`-loop that performs calculations for a preset number of repetitions, a `while`-loop allows you to perform calculations as long as a certain condition is true. For instance, you could set up a `while`-loop that runs as long as it is `true` that the current position is not on the edge of the domain.

▶ For which value(s) of `CurrentRow` and `CurrentCol` in `valleydem` can you apply your algorithm? Complete Table 12.1.

Table 12.1: While loop conditions

| **Condition** | **Relational expression in MATLAB** |
|---|---|
| `CurrentRow` is greater than 1 | ... |
| ... | ... |
| ... | ... |
| ... | ... |

▶ In your script m-file, replace the `for`-loop with a `while`-loop that combines the conditions as stated above.

▶ Load 'valleydem2.txt' and visualize it. Can you see the difference between `valleydem` and `valleydem2`?

▶ Try to use `valleydem2` in your drainage program (if needed you can stop your program by simultaneously pressing the 'Ctrl' and 'c' buttons while clicking in the MATLAB command window.

As you probably found out for yourself, you end up with a never-ending loop because the new DEM contains so-called pits: local minimums. The track ends in a cell with only higher cells surrounding it. You can overcome this by adding an extra condition in your while-statement that tests if the current position is in a pit or not.

▶ Alter the expression in your while loop in such a way that it tests for pits.

---

End of Project 7.

## Project 8. Drainage pattern

- ► Copy your script m-file from the previous project to the directory '\chap12_more_projects\proj07_drainpattern' and reset your work folder accordingly. Change the script m-file into a function m-file '∗*name*∗_draintrack.m' (with your last name for ∗*name*∗), with `StartRow`, `StartCol` and `DEM` as input arguments and `Track` as output:

```
function Track = draintrack(DEM,StartRow,StartCol)
```

  To get an impression of the total drainage pattern of this valley system, you must run the algorithm for every possible position in `valleydem`. In order to do this, follow the outline below:

  1. Write a script m-file ('∗*name*∗_drainpattern.m'), with a double `for`-loop that calls `draintrack`.
  2. In your `for`-loop, take increments of 10 to avoid long calculation times.
  3. Most of the initial part and the visualization is now not needed in the function and can thus be transported to the main program. Cut and paste as much as possible from the `draintrack` function to the main program.
  4. Check whether the program works well. Use the debugger if necessary.

- ► Now also try to use the Luxembourg DEM ('demlux.txt') instead of `valleydem`.

  As you can see from the error, an analytical error still exists in our program: during calculation, some operations can not be executed because of incorrect variable dimensions. This happens when the program has to deal with more than one lowest gradient and doesn't know which one to use. A relatively easy way to solve this is by always taking the first point.

- ► Alter your program to always use the first point, even when there are multiple directions with the same lowest gradient. The end result should look like Figure 12.5.

- ► The function m-file 'randchoosefrom.m' that is present in your directory, lets you pick a value at random if there is more than one direction of steepest descent. Open the function, read the help block, and incorporate it in your script. Note that it's easy to make an analytical mistake here, which will cause your program to calculate an incorrect result (even though it won't raise an error). So think first about what you need to do, before you start implementing.

- ► Drainage tracks are shorter when the terrain is relatively flat. Why?

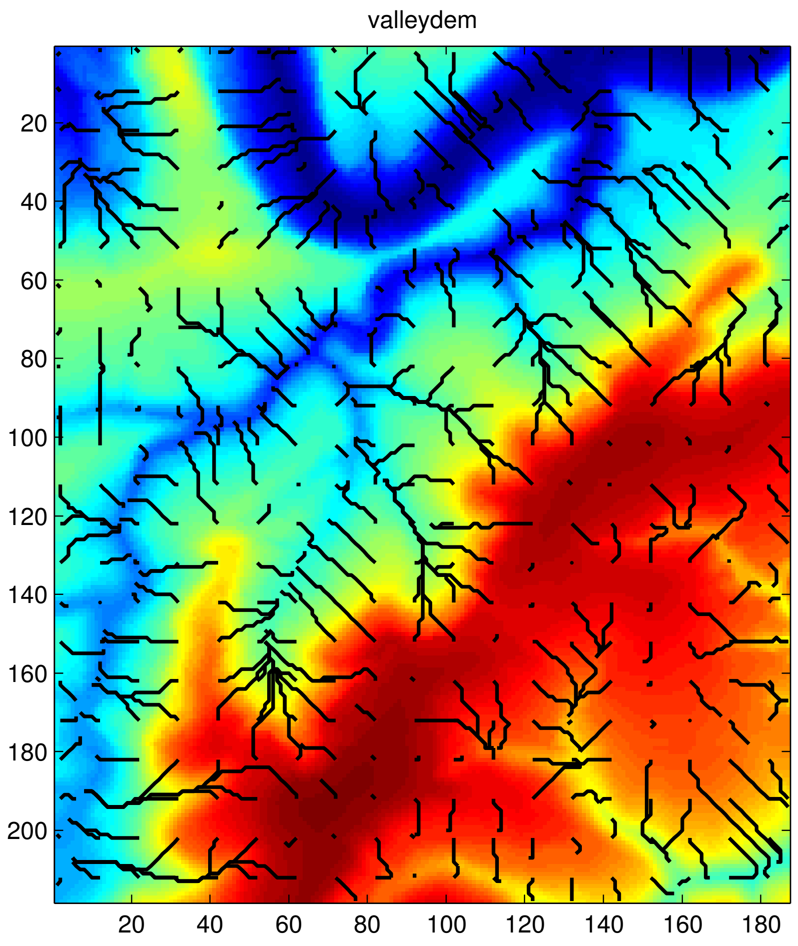


Figure 12.5: Drainage pattern of the Luxembourg DEM, as determined by the steepest-descent method.

Chapter 13

# External toolboxes

## 13.1 User contributed software

Besides the algorithms that come with the MATLAB program, there is a wealth of user-contributed programs available on the Internet. A good place to look for such programs, is the 'File Exchange' on the Mathworks website http://www.mathworks.com/matlabcentral/fileexchange/?sort=downloads.

▶ Take 20 minutes to browse through the contents of the File Exchange website. On the left side of the website, there is an item 'Files by Category' that you may want to use to limit the search to the 'Earth Sciences' domain.

As you can see, there are many programs available that you can use for free. In this chapter we will use the GoogleEarth Toolbox for MATLAB: http://www.mathworks.com/matlabcentral/fileexchange/12954-google-earth-toolbox. You have a copy of the GoogleEarth Toolbox, located in the folder 'pim_files\'. In this chapter, you will learn to use the functions in this toolbox.

---

Project 9. Google Earth Toolbox

---

## 13.2 Installing GoogleEarth

▶ Open Windows Explorer and navigate to the folder 'pim_files\ch13_toolboxes'. There should be a file 'GoogleEarthPerUserSetup.exe'. Click to install the GoogleEarth Viewer.

## 13.3 Setting up the GoogleEarth application

To avoid problems with displaying and interpreting your GoogleEarth files, it's necessary to have your copy of GoogleEarth set up according to the following specifications:

1. Click “Tools” in GoogleEarth’s menu and choose “Options…”. On the tab named “3D View” check that “Graphics Mode” is set to “OpenGL”.
2. On the same tab, set the latitude/longitude format to “Decimal Degrees” under “Show Lat/Long”.
3. Also on this tab, set the units of elevation to “Meters, Kilometers” under “Show Elevation”.

## 13.4 KML

The great flexibility of GoogleEarth stems from its use of XML[1]-based text files, known as KML[2]-files (you can recognize these files by their extension *.kml). Such a file typically contains a series of objects, such as polygons, lines and points. Each of these objects is represented within a KML-file by its KML-tags. For example, we may find a polygon object:

```
<Polygon>
...
</Polygon>
```

Or a line object:

```
<LinearRing>
...
</LinearRing>
```

Properties such as line width, coordinates, and polygon color may be specified using additional tags. A line may thus be assigned a line style as follows:

```
<LineStyle>
   <color>ffffffff</color>
   <width>1.00</width>
</LineStyle>
```

Which represents a fully opaque, white line of width 1.

### 13.4.1 Automated writing of KML

Of course, writing KML-tags by manually typing them in a text editor is not practical when working with data containing more than just a few objects. Fortunately though, the task of writing these files can be fully automated using the GoogleEarth Toolbox.

The GoogleEarth Toolbox generates plain ASCII text files. The text contained within these files is formatted according to the Keyhole Markup Language (KML) specification,

[1] http://en.wikipedia.org/wiki/XML
[2] http://en.wikipedia.org/wiki/Keyhole_Markup_Language

which allows for a hierarchical organization of the objects and properties contained within the file. KML files can be opened in the GoogleEarth Viewer.

The GoogleEarth Toolbox implements a number of low-level functions for displaying point, line and polygon objects within GoogleEarth. In addition to that, higher-level methods include functions to visualize 2D rasterized data, create contour maps, surface and quiver plots, image overlays (in GIS usually referred to as *draping*), and a few others.

## 13.5 Structure of the GoogleEarth Toolbox

The core of the toolbox is formed by a set of about 20 MATLAB functions (*.m files) located in the toolbox root folder ('googleearth/'). The majority of these functions generate character arrays in concordance with the KML standard. For each of the m-files, a help file has been included which can be viewed from within the MATLAB help browser. The help files are located in 'googleearth/html'. In addition to the help files, demonstrations have been included ('googleearth/demo'). Many of these demos write their output to a separate folder ('googleearth/kml') in order to avoid contaminating directories with files that do not belong there. Folder 'googleearth/data' contains additional files such as icon images and Collada models. Folder 'googleearth/doc' contains files pertaining to the printable documentation. Finally, a folder 'googleearth/tmp' has been included, which is used by some functions to write temporary data to. The contents of this folder are automatically emptied by some functions, therefore you should not save important data in it. In fact, it's best if you don't save any file anywhere under 'googleearth/'.

## 13.6 Adding the toolbox to the path

When you use a function name in your scripts or at the prompt, MATLAB starts searching a predefined list (i.e. the 'path') of directories where MATLAB functions are stored.

► At the prompt, type

```
>> path
```

The default list of directories should now be displayed in the command window. In order to use the functions from the GoogleEarth Toolbox, you need to add the toolbox' top-level directory to the path:

```
>> addpath('D:\matlab\googleearth')
```

This will add the indicated directory to the MATLAB search path (see `doc path`). The above command is equivalent to using "Set Path..." from the "File" menu of the MATLAB Graphical User Interface.

► Now type

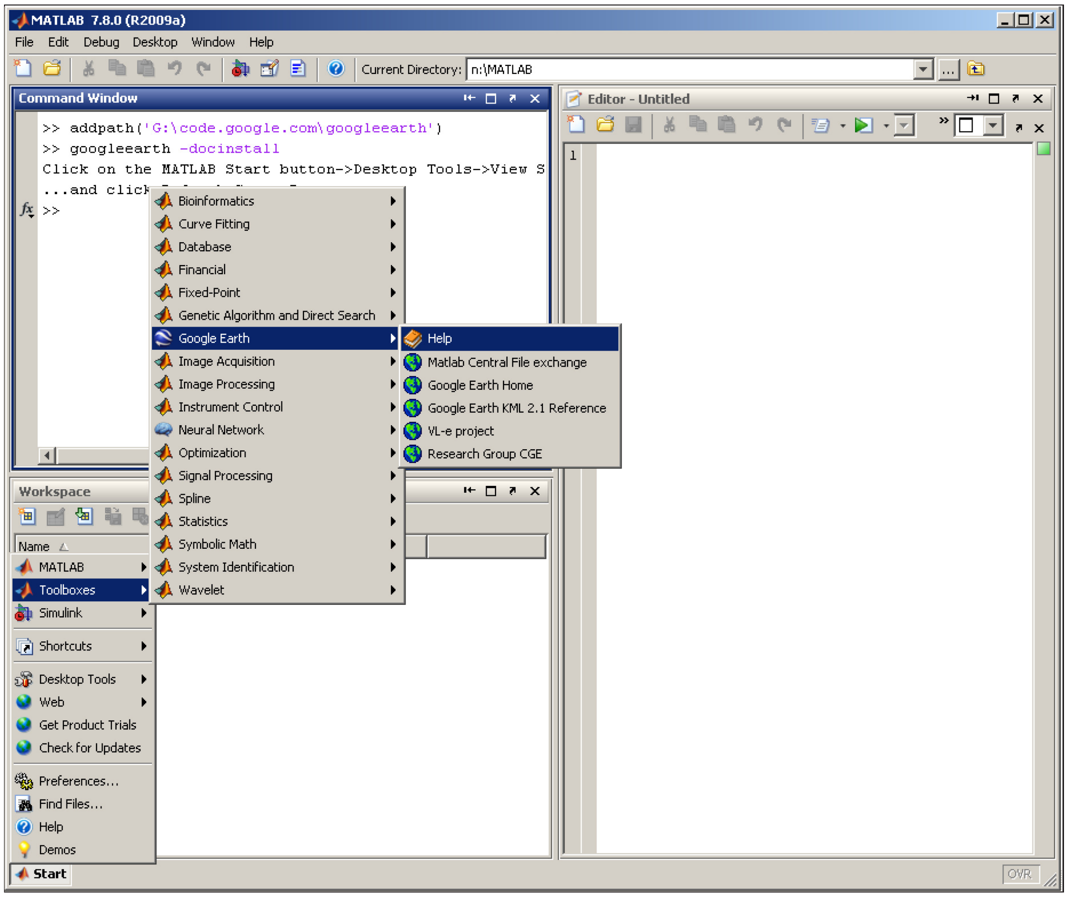
```
>> googleearth -docinstall
```

to initialize some files that are needed to use the documentation.

After adding the toolbox location to the MATLAB path, its functions may be called either from the command window or in scripts. Additionally, the toolbox documentation is accessible through the usual `doc` command. Typing

```
>> doc googleearth
```

and following the hyperlink will take you to an index page where all functions are listed, along with a short description of their intended usage. As an alternative to using the `doc` command, the documentation may also be accessed by selecting the GoogleEarth item which is now located under MATLAB's Start button ➝ Toolboxes (see Figure 13.1). In the documentation files, you can find information on what a particular function does, what variables to pass it, the function's options, links to other functions that perform related tasks, and an example of how the function can be used.



```
>> addpath('G:\code.google.com\googleearth')
>> googleearth -docinstall
Click on the MATLAB Start button->Desktop Tools->View S
...and click
>>
```

Figure 13.1: Accessing the GoogleEarth Toolbox documentation.

## 13.7 Point features

▶ In the GoogleEarth Viewer, go to a location of your choice and write down its latitude and longitude.

▶ Navigate to the documentation on `ge_point` or enter the command below at the prompt:

```
>> doc ge_point
```

This should invoke the documentation about `ge_point`. Read through it carefully.

- ▶ Begin a new m-file called '∗*name*∗_point.m'. Let the script generate a character array called `kmlStr`, which contains the location of your choice in KML format.

  You should now have a character array with KML-formatted text. However, to be able to see your point in the GoogleEarth Viewer, we need to write the character array to a *.kml file.

- ▶ Consult the documentation on `ge_output`, and use it to save your KML-formatted text to a *.kml file.

- ▶ Open your *.kml file in the GoogleEarth Viewer and verify that it shows a point at the location of your choice.

- ▶ By default, the point is labeled with the full filename of its *.kml file when opened in the GoogleEarth Viewer (see left-hand pane "Places"). It's often useful to assign a different name to the object when viewed in GoogleEarth using `ge_output`'s parameter `'name'`. Change how your point is displayed in the "Places" pane in the GoogleEarth Viewer.

- ▶ Instead of the default placemarker icon, use this `'iconURL'` (or any other icon of your choice): `http://maps.google.com/mapfiles/kml/pal3/icon35.png`.

## 13.8 Line features

Lesser black backed gulls (*Larus fuscus*) are sea birds that spend spring and summer in The Netherlands but, like some of its human inhabitants, spend their winters in Southern Europe. While migration patterns of sea birds have been monitored before, until now it was not possible to get the detailed information about their movements needed to gain insight into their foraging behavior. Therefore, the University of Amsterdam developed a new flexible tracking system that enables detailed monitoring of the bird's movement. These so-called 'GPS-loggers' consist of a tiny GPS system combined with a miniature recording device and transceiver, powered with a solar panel. They weigh only 18 grams, allowing birds to carry them on their backs without being restricted in their movements (see Figure 13.2). Once attached to the birds, the devices record their position, wing beat frequency, temperature and air pressure during their local and migratory movements with great detail. In this section, you will use the GoogleEarth Toolbox to visualize the migratory route that gull #41745 has taken.

- ▶ Start a new script '∗*name*∗_seagull.m'.

- ▶ Use `textread` to load the seagull data from the file 'pim_files\ch13_toolboxes\larus-fuscus.txt'. If you are not successful doing this but there's no one around to help you, use `load('larus-fuscus-cheat.mat')` to read from a binary file, so that you won't waste too much time. (But don't forget to ask about `textread` later).

- ▶ Consult the documentation on the use of `ge_plot`, and use it to visualize the bird's track. Adapt the script so that it produces a fully opaque, red line of width 1.5. If you do this correctly, the data should be displayed as in Figure 13.3.

Figure 13.2: Lesser black-backed gull (*Larus fuscus*) equipped with a GPS logger (Photo: Cees Camphuysen, NIOZ).

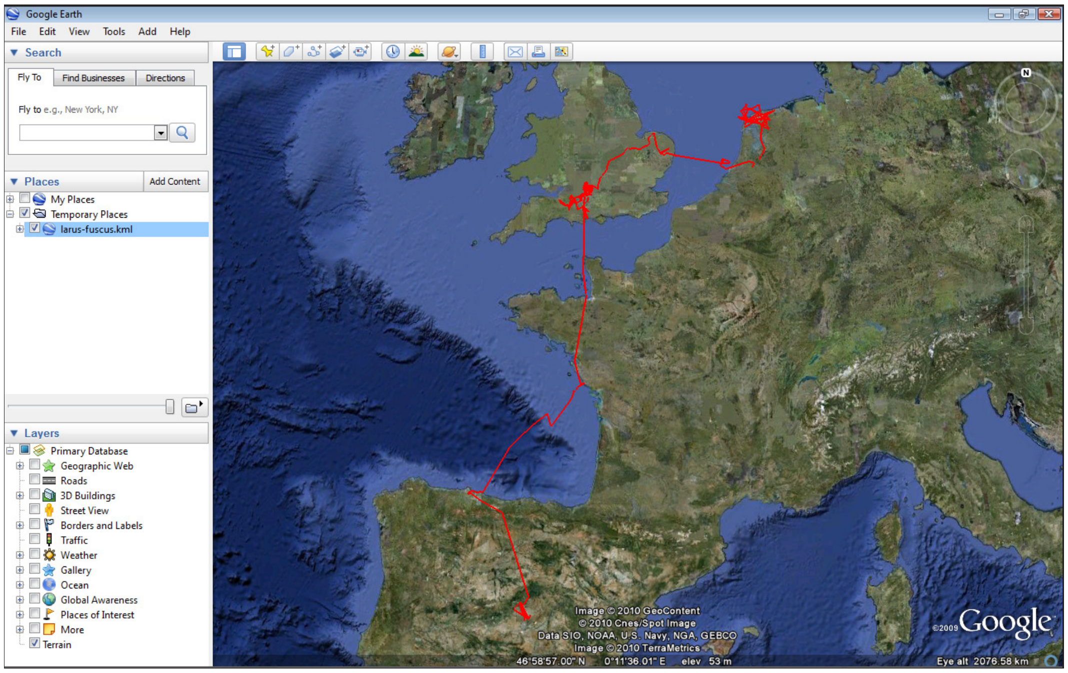


Figure 13.3: Bird track of seagull #41745.

▶ Alter your script in such a way that it displays a point for each location where a measurement was taken. For this, you need to iterate over the locations, generate a `ge_point`, and concatenate the result with the KML-text that you already had. You can use any icon that you like (but `http://maps.google.com/mapfiles/kml/pal2/icon26.png` is a good option, especially when used in combination with the `'iconScale'` and `'iconColor'` options). If you open your *.kml file in the GoogleEarth Viewer, you should see the red line of Figure 13.3, as well as many points.

So far, you have generated KML files that contain objects which are static, i.e. they do not change with time. However, sometimes you want to present your data in a dynamic way, somewhat like a movie. The KML specification allows for the inclusion of objects which are visible only during a particular time interval. To make this work, the object in question (such as a point or a line) needs to be assigned a start time and an end time, between which it will be visible. The start and stop times need to be passed as character arrays using the option `'timeSpanStart'` and `'timeSpanStop'`. The character array should be formatted according to the following layout: `2008-01-05T22:00:00Z` , or 10 PM on the $5^{th}$ of January 2008.

▶ Alter your `textread` call in such a way that, in addition to the latitude and longitude, the time information is now also loaded into the workspace as two separate variables `startDateTime` and `endDateTime`.

▶ Use a `for`-loop to iterate over all measurements. For each measurement, generate a KML string using `ge_point` with the timespan options. Concatenate and write to a *.kml file as before. When opened in the GoogleEarth Viewer, you should see a line representing the entire migratory route of the gull, as well as a point representing its location at a particular time. When played back, the point should move along the migratory route.

---

**TIP:** When the objects in the GoogleEarth Viewer are dynamic (i.e. when they have time labels), the upper left part of the main window contains additional controls with buttons for playback of the KML objects.

---

## 13.9 Rasterized imagery

In the domain of Earth Sciences and Biology, we often have data in the form of maps. In this section, you will use the GoogleEarth Toolbox to visualize a map of Zinc concentrations in the Meuse river valley near Stein, The Netherlands. An array with spatial predictions of Zinc has already been derived from point measurements by means of a geostatistical method (ordinary kriging). This data is available as 'pim_files\ch13_toolboxes\ok-predictions-radius.mat'.

▶ Start a new script '∗*name*∗_zinc.m'. Let it load the kriging data[3] and visualize it using

[3]Data from: Rikken, M. G. J. and Van Rijn, R. P. G. (1993) *Soil pollution with heavy metals –An inquiry into spatial variation, cost of mapping and the risk evaluation of copper, cadmium, lead, and zinc in the floodplains of the Meuse west of Stein, the Netherlands.* Dept. of Physical Geography, Utrecht University

imagesc(pre). Include a title and a colorbar.

- ▶ Consult the documentation on how to visualize raster-based images using `ge_imagesc`.

Table 13.1 contains the boundaries of the map. As you can see, they are given in degrees-minutes-seconds notation; for instance, the Northern border is at 50 degrees, 59 minutes, 31.20 seconds. Since a minute is 1/60 of a degree, and a second is 1/3600 of a degree, the top border is at 50.9920 decimal degrees. The toolbox only accepts latitudes and longitudes in decimal degrees.

- ▶ You must write a function that converts degrees-minutes-seconds to decimal degrees. Start a new function m-file. Save it as '∗*name*∗_degminsec2decdeg'. The function should have 3 numerical input arguments (degrees, minutes and seconds), and one output argument (decimal degrees). Don't forget to include a comment help block.

Table 13.1: Zinc map boundaries

| **Boundary** | **Location** |
|---|---|
| North | 50°59'31.20" |
| East | 5°46'11.40" |
| South | 50°57'18.91" |
| West | 5°43'13.36" |

- ▶ For negative latitudes and longitudes, the function probably does not give the right result (-50°59'31.2" ≠ -49.0080°). Alter your function in such a way that it first checks whether the input argument 'degrees' is negative, and `if` that is the case, make a slightly different calculation.

- ▶ Study Code Snippet 13.1 (You can find a copy of this file in your work directory 'pim_files\ch13_toolboxes\'). After studying the example m-file, use `ge_imagesc` to let your script generate a KML-formatted character array and have it written to disk.

- ▶ Open your *.kml file in the GoogleEarth Viewer and verify that it is displayed correctly.

- ▶ Consult the documentation on `ge_colorbar`. Let your script generate the KML-formatted text that contains the colorbar information. Concatenate the 2 KML character arrays and pass them to `ge_output`.

- ▶ Go back to the GoogleEarth Viewer, and right-click on the name of your object. Select "Revert" to let GoogleEarth reload the file. Check that a colorbar is now included in the file.

- ▶ By default, the colorbar is labeled 'ge_colorbar' when displayed in the GoogleEarth Viewer. Change it to a more descriptive name.

- ▶ What is the lowest value in `pre`? This value is used to determine the range of colors, even though the map does not contain any concentrations below 119.368 ppm. As a result of the stretched-out colorbar, there is not enough color differentiation in the research area. Let your script remedy this by setting the color limits explicitly.

Code Snippet 13.1: Contents of the file 'example_ge_imagesc.m'

```
clear        % clear the workspace
close all    % close any open figures
clc          % clean up the command window

% initialize the longitude vector:
lonVec = [3,4,5];

% initialize the latitude vector:
latVec = [32.5:0.25:33.25];

% initialize the data:
matrix4x3 = [1,2,3;4,5,6;7,8,9;10,11,12];

% generate KML formatted text:
kmlStr = ge_imagesc(lonVec,latVec,matrix4x3);

% write the KML to file:
ge_output('example_ge_imagesc.kml',kmlStr)

% the kml file should display similar to this:
figure
imagesc(lonVec,latVec,flipud(matrix4x3))
xlabel('longitude')
ylabel('latitude')
title('example of imagesc')
colorbar
set(gca,'ydir','normal') % make sure the y-axis increases
                         % upward, instead of downward.
```

- ▶ In the GoogleEarth Viewer, verify that the colors on the colorbar no longer correspond with those that are displayed. Change your script in such a way that the color limits are also applied to the colorbar.

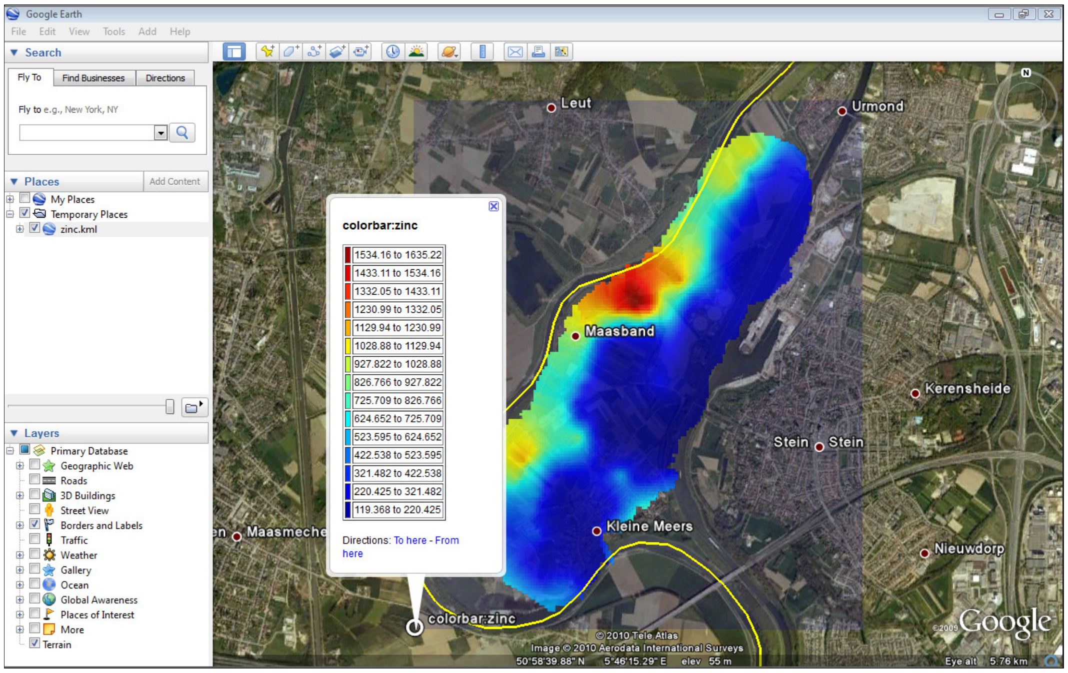
- ▶ The array `pre` contains -1 values where the kriging method has not been applied. Using `ge_imagesc`'s option `'alphaMatrix'`, you can specify for each element in `pre` how transparent it should be. Use relational operators to make an array `transpArray` that contains the transparency information. Consult the documentation on the option `'alphaMatrix'` to make the area surrounding the research area 80% transparent.

- ▶ Make the research area itself 20% transparent.

If you have executed all the exercises correctly, your *.kml file should look like Figure 13.4.

---

End of Project 9.



colorbar:zinc

| | |
|---|---|
| 🟥 | 1534.16 to 1635.22 |
| 🟥 | 1433.11 to 1534.16 |
| 🟥 | 1332.05 to 1433.11 |
| 🟧 | 1230.99 to 1332.05 |
| 🟧 | 1129.94 to 1230.99 |
| 🟨 | 1028.88 to 1129.94 |
| 🟩 | 927.822 to 1028.88 |
| 🟩 | 826.766 to 927.822 |
| 🟩 | 725.709 to 826.766 |
| 🟦 | 624.652 to 725.709 |
| 🟦 | 523.595 to 624.652 |
| 🟦 | 422.538 to 523.595 |
| 🟦 | 321.482 to 422.538 |
| 🟦 | 220.425 to 321.482 |
| 🟦 | 119.368 to 220.425 |

Directions: To here - From here

Figure 13.4: Zinc concentrations near Stein

# Index