An Introduction to MATLAB for Geoscientists

Dave Heslop

Online Edition: November 2012

Contents

Chapter 1

Introduction

1.1 MATLAB

MATLAB, short for *Matrix Laboratory*, is a numerical computing environment and programming language originally developed by Cleve Moler. Evolving dramatically from its first release, MATLAB is now over 30 years old and is distributed commercially by The MathWorks Inc. In this course we will work through a number examples which are designed to help you come to terms with the basics of MATLAB and see how you can use it in your own research. Because MATLAB is programmable it provides an environment in which we can process scientific data without the need to use specific software packages which are limited in the tasks they can perform. The three main advantages of using MATLAB for your day-to-day scientific work are:

- It is an interpreted language, this means we can work step by step through a series of commands without the need for compiling complete sets of code.
- MATLAB is based around so-called "inbuilt" functions which provide quick access to thousands of different data processing methods.
- MATLAB contains versatile graphics libraries allowing you to view your data in a variety of ways as it is processed.

One of the first things you'll notice about MATLAB is that it is command driven. If you are normally working with EXCEL you probably spend 95% of your time using the mouse and only 5% typing on the keyboard. In MATLAB the opposite is true and whilst this might seem annoying at first, you'll hopefully soon come to recognize why this is necessary.

1.1.1 Obtaining a copy of MATLAB

In its distribution of MATLAB The MathWorks promotes the a system based around "toolboxes". This means that MATLAB by itself is relatively cheap, but if you want to perform specialist tasks such as statistics, image processing or optimization you can also buy specific toolboxes which greatly enhance the methods available to you. Currently there are over fifty different toolboxes available for tasks as diverse as *modelling biological systems* to *fuzzy logic* and you have to pay about 400 euros for each one you use. Overall this means MATLAB can be very expensive once you add on a number of toolboxes (although free versions of some toolboxes have been written and are available over the internet). Of course if you are only using MATLAB at work then you can rely on the university's license, otherwise you can look into the student version of MATLAB which includes 7 toolboxes and costs only ∼100 euros.

1.1.2 Useful MATLAB books

It is estimated that over one million people use MATLAB so it is not surprising that there are a wide variety of textbooks available (some very good and some very bad) which discuss MATLAB in a number of contexts. Some good quality MATLAB books are available for free and can be downloaded in pdf form. A good place to start is The MathWorks web site, which contains a specific section for books including a subsection for the "Earth Sciences". Two very good books, by Cleve Moler, that you can download for free are Numerical Computing with MATLAB and Experiments with MATLAB.

If you are going to buy a single MATLAB book I would recommend "Mastering MATLAB 7" by Hanselman & Littlefield. This book provides an introduction to nearly all aspects of MATLAB and it is relatively cheap (the library also has a number of copies). There are also two books available which show how MATLAB can be applied to problems in the Earth Sciences (both of these books are also in the library):

- *Data Analysis in the Earth Sciences using MATLAB* by Middleton
- *MATLAB Recipes for Earth Sciences* by Trauth

1.1.3 Everyday help with MATLAB

The last thing to mention before we get started is that MATLAB has an amazingly detailed help system which you can access from the software directly or over the internet. This is an essential reference for anyone working with MATLAB with the bonus that worked-examples are provided to show you how to perform specific tasks. MATLAB even has its own You-Tube channel where people post videos which act as tutorials on a variety of different problems. Finally, if you are not feeling motivated enough to visit the library you'll find that spending a few minutes on the internet will provide you with an almost unlimited number of free tutorials and online classes. I typed "MATLAB tutorial" into Google and got over 250,000 hits!

1.1.4 How these notes are structured

In writing these notes I have attempt to demonstrate the workings of MATLAB using examples. Throughout the notes your see the MATLAB examples given in a different font, for example:

```
This is the MATLAB example font
```

You can follow the examples by typing in the listed commands. There are two important things to keep in mind when following the examples. First, this is more than a typing exercise, it is essential that you try to understand what the commands are doing and how they work. Second, if you get an error it is probably because you have made a typing mistake, so **check your typed commands carefully**. You might also notice that there are slight differences between the example commands given in the on-screen presentation and in this document. This is simply a space issue, many of the comments in the on-screen presentation are shortened so that they will fit on a single line, in this document more extensive comments can be given.

1.2 Getting started

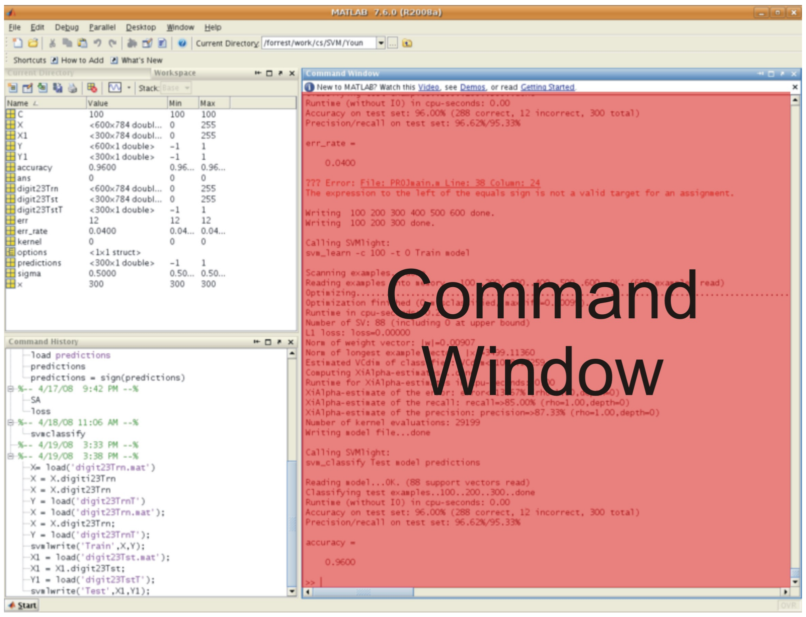
Find the MATLAB icon on your desktop, double click it and wait for MATLAB to start (this may take a little time). The MATLAB desktop is quite complicated so we'll examine it step by step.

1.2.1 The Command Window

Most of your interaction with MATLAB will be performed by typing commands into the command window. We will issue commands at the MATLAB prompt, which looks like `>>`. You can even try giving a command at the prompt, for example:

```
>> why
```

then press the *enter* key. You should get a slightly strange message returned on the screen.


MATLAB 7.6.0 (R2008a)
File Edit Debug Parallel Desktop Window Help
Current Directory: /forrest/work/cs/SVM/Youn
Shortcuts How to Add What's New
Current Directory Workspace
Stack: Base
Command History
Command Window
New to MATLAB? Watch this Video, see Demos, or read Getting Started.
Command Window
Start
OVR

Name	Value	Min	Max
C	100	100	100
X	<600x784 doubl...	0	255
X1	<300x784 doubl...	0	255
Y	<600x1 double>	-1	1
Y1	<300x1 double>	-1	1
accuracy	0.9600	0.96...	0.96...
ans	0	0	0
digit23Trn	<600x784 doubl...	0	255
digit23Tst	<300x784 doubl...	0	255
digit23TstT	<300x1 double>	-1	1
err	12	12	12
err_rate	0.0400	0.04...	0.04...
kernel	0	0	0
options	<1x1 struct>		
predictions	<300x1 double>	-1	1
sigma	0.5000	0.50...	0.50...
x	300	300	300

```
  load predictions
  predictions
  predictions = sign(predictions)
%-- 4/17/08  9:42 PM --%
  SA
  loss
%-- 4/18/08 11:06 AM --%
  svmclassify
%-- 4/19/08  3:33 PM --%
%-- 4/19/08  3:38 PM --%
  X= load('digit23Trn.mat')
  X = X.digiti23Trn
  X = X.digit23Trn
  Y = load('digit23TrnT')
  X = load('digit23Trn.mat');
  X = X.digit23Trn;
  Y = load('digit23TrnT');
  svmlwrite('Train',X,Y);
  X1 = load('digit23Tst.mat');
  X1 = X1.digit23Tst;
  Y1 = load('digit23TstT');
  svmlwrite('Test',X1,Y1);
```

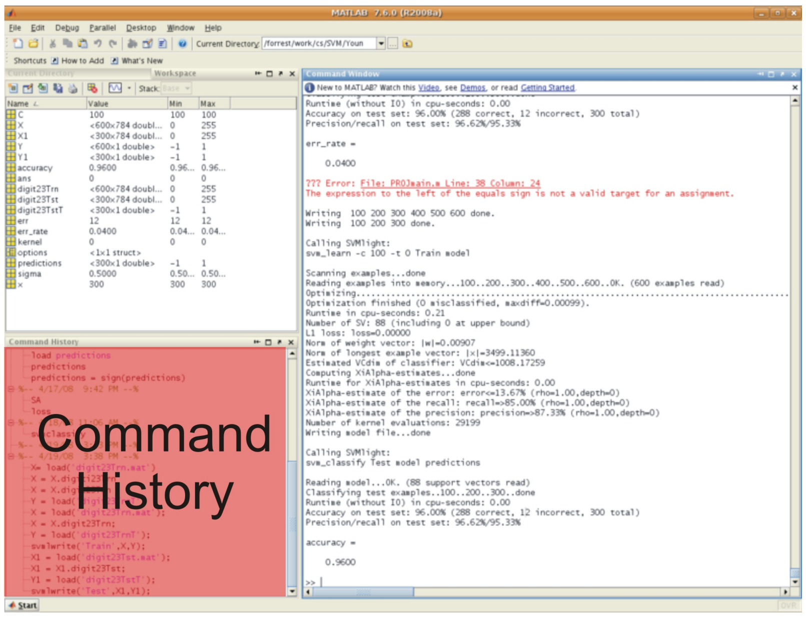
```
Runtime (without IO) in cpu-seconds: 0.00
Accuracy on test set: 96.00% (288 correct, 12 incorrect, 300 total)
Precision/recall on test set: 96.62%/95.33%

err_rate =

    0.0400

??? Error: File: PROJmain.m Line: 38 Column: 24
The expression to the left of the equals sign is not a valid target for an assignment.

Writing  100 200 300 400 500 600 done.
Writing  100 200 300 done.

Calling SVMlight:
svm_learn -c 100 -t 0 Train model

Scanning examples.[illegible]
Reading examples into memory...100..200..300..400..500..600..OK. (600 examples read)
Optimizing.............................................................................
Optimization fin[illegible]hed ([illegible]classified, max[illegible]f=0.[illegible]).
Runtime in cpu-sec[illegible]
Number of SV: 88 (including 0 at upper bound)
L1 loss: loss=0.00000
Norm of weight vector: |w|=0.00907
Norm of longest example [illegible] |x[illegible]499.11360
Estimated VCdim of class[illegible]VCdim<[illegible]259
Computing XiAlpha-estimat[illegible]
Runtime for XiAlpha-esti[illegible]pu-[illegible]
XiAlpha-estimate of the er[illegible]or<[illegible]
XiAlpha-estimate of the recall: recall=>85.00% (rho=1.00,depth=0)
XiAlpha-estimate of the precision: precision=>87.33% (rho=1.00,depth=0)
Number of kernel evaluations: 29199
Writing model file...done

Calling SVMlight:
svm_classify Test model predictions

Reading model...OK. (88 support vectors read)
Classifying test examples..100..200..300..done
Runtime (without IO) in cpu-seconds: 0.00
Accuracy on test set: 96.00% (288 correct, 12 incorrect, 300 total)
Precision/recall on test set: 96.62%/95.33%

accuracy =

    0.9600

>> |
```

Figure 1.1: *The main MATLAB screen with the Command Window highlighted in red.*

1.2.2 The Command History

This is one of the particularly useful features of MATLAB which will save you a lot of typing. The Command History provides a sequential list of all the type commands that have been entered into the Command Window. This means you can recall a single command or a collection of commands and re-execute them with a click of the mouse. If you tried the `why` command above you will see that it will now have appeared as the bottom item of the Command History. Just double-click this command in the Command History and it will be re-executed in the Command Window (and you will get a different but equally bizarre reply). The Command

History list can also be accessed from the Command Window itself, if you place your cursor at the >> prompt and start pressing the cursor, MATLAB will list backwards through the previous commands. Once you have found the command you want, you can execute it by hitting the *enter* key.


MATLAB 7.6.0 (R2008a)
File Edit Debug Parallel Desktop Window Help
Current Directory: /forrest/work/cs/SVM/Youn
Shortcuts How to Add What's New
Current Directory
Workspace
Stack: Base
Command History
Command Window
New to MATLAB? Watch this Video, see Demos, or read Getting Started.
Command History
Start
OVR

Name	Value	Min	Max
C	100	100	100
X	<600x784 doubl...	0	255
X1	<300x784 doubl...	0	255
Y	<600x1 double>	-1	1
Y1	<300x1 double>	-1	1
accuracy	0.9600	0.96...	0.96...
ans	0	0	0
digit23Trn	<600x784 doubl...	0	255
digit23Tst	<300x784 doubl...	0	255
digit23TstT	<300x1 double>	-1	1
err	12	12	12
err_rate	0.0400	0.04...	0.04...
kernel	0	0	0
options	<1x1 struct>		
predictions	<300x1 double>	-1	1
sigma	0.5000	0.50...	0.50...
x	300	300	300

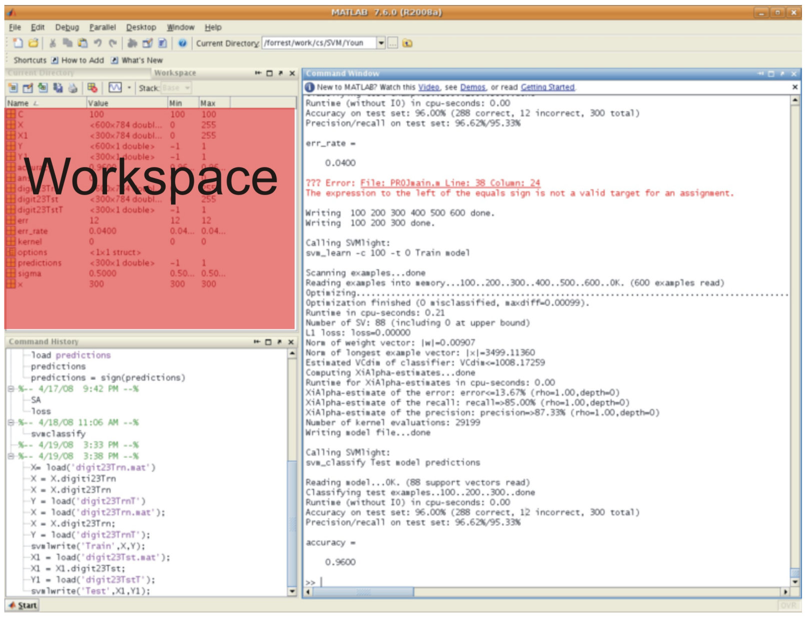
```
load predictions
predictions
predictions = sign(predictions)
%-- 4/17/08  9:42 PM --%
SA
loss
%-- 4/18/08 11:06 AM --%
svmclassify
%-- 4/19/08  3:33 PM --%
%-- 4/19/08  3:38 PM --%
X= load('digit23Trn.mat')
X = X.digiti23Trn
X = X.digit23Trn
Y = load('digit23TrnT')
X = load('digit23Trn.mat');
X = X.digit23Trn;
Y = load('digit23TrnT');
svmlwrite('Train',X,Y);
X1 = load('digit23Tst.mat');
X1 = X1.digit23Tst;
Y1 = load('digit23TstT');
svmlwrite('Test',X1,Y1);
```

```
Runtime (without IO) in cpu-seconds: 0.00
Accuracy on test set: 96.00% (288 correct, 12 incorrect, 300 total)
Precision/recall on test set: 96.62%/95.33%

err_rate =

    0.0400

??? Error: File: PROJmain.m Line: 38 Column: 24
The expression to the left of the equals sign is not a valid target for an assignment.

Writing  100 200 300 400 500 600 done.
Writing  100 200 300 done.

Calling SVMlight:
svm_learn -c 100 -t 0 Train model

Scanning examples...done
Reading examples into memory...100..200..300..400..500..600..OK. (600 examples read)
Optimizing.............................................................................................
Optimization finished (0 misclassified, maxdiff=0.00099).
Runtime in cpu-seconds: 0.21
Number of SV: 88 (including 0 at upper bound)
L1 loss: loss=0.00000
Norm of weight vector: |w|=0.00907
Norm of longest example vector: |x|=3499.11360
Estimated VCdim of classifier: VCdim<=1008.17259
Computing XiAlpha-estimates...done
Runtime for XiAlpha-estimates in cpu-seconds: 0.00
XiAlpha-estimate of the error: error<=13.67% (rho=1.00,depth=0)
XiAlpha-estimate of the recall: recall=>85.00% (rho=1.00,depth=0)
XiAlpha-estimate of the precision: precision=>87.33% (rho=1.00,depth=0)
Number of kernel evaluations: 29199
Writing model file...done

Calling SVMlight:
svm_classify Test model predictions

Reading model...OK. (88 support vectors read)
Classifying test examples..100..200..300..done
Runtime (without IO) in cpu-seconds: 0.00
Accuracy on test set: 96.00% (288 correct, 12 incorrect, 300 total)
Precision/recall on test set: 96.62%/95.33%

accuracy =

    0.9600

>>
```

Figure 1.2: *The main MATLAB screen with the Command History highlighted in red.*

1.2.3 Workspace

This window is located in the top right corner of the screen (make sure that the *Workspace* tab is selected rather than the *Current Directory*). Once we start to work with MATLAB this window will keep a record of the information which is stored in MATLAB's memory.


MATLAB 7.6.0 (R2008a)
File Edit Debug Parallel Desktop Window Help
Current Directory: /forrest/work/cs/SVM/Youn
Shortcuts How to Add What's New
Current Directory
Workspace
Stack: Base
Workspace
Command History
Command Window
New to MATLAB? Watch this Video, see Demos, or read Getting Started.
Start
OVR

Name	Value	Min	Max
C	100	100	100
X	<600x784 doubl...	0	255
X1	<300x784 doubl...	0	255
Y	<600x1 double>	-1	1
Y1	<300x1 double>	-1	1
accuracy	0.9600	0.96...	0.96...
ans	0	0	0
digit23Trn	<600x784 doubl...	0	255
digit23Tst	<300x784 doubl...	0	255
digit23TstT	<300x1 double>	-1	1
err	12	12	12
err_rate	0.0400	0.04...	0.04...
kernel	0	0	0
options	<1x1 struct>		
predictions	<300x1 double>	-1	1
sigma	0.5000	0.50...	0.50...
x	300	300	300

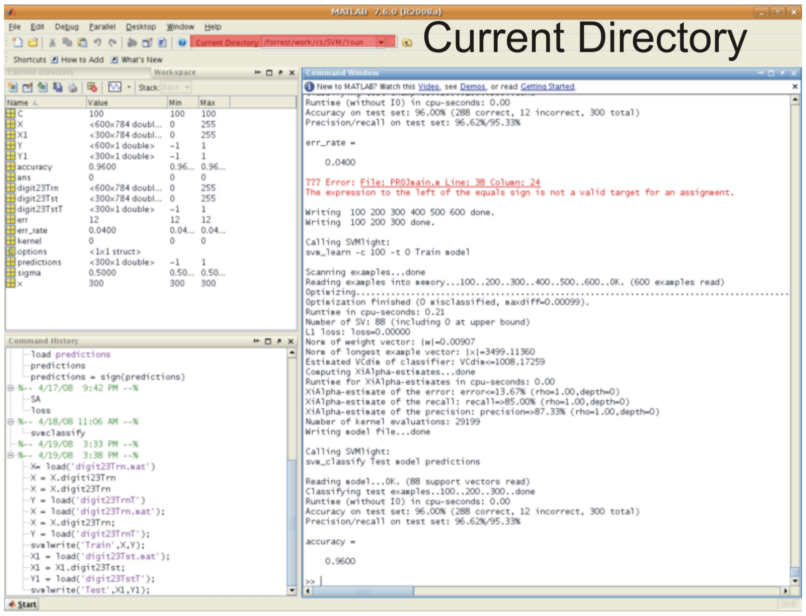
```
load predictions
predictions
predictions = sign(predictions)
%-- 4/17/08  9:42 PM --%
SA
loss
%-- 4/18/08 11:06 AM --%
svmclassify
%-- 4/19/08  3:33 PM --%
%-- 4/19/08  3:38 PM --%
X= load('digit23Trn.mat')
X = X.digiti23Trn
X = X.digit23Trn
Y = load('digit23TrnT')
X = load('digit23Trn.mat');
X = X.digit23Trn;
Y = load('digit23TrnT');
svmlwrite('Train',X,Y);
X1 = load('digit23Tst.mat');
X1 = X1.digit23Tst;
Y1 = load('digit23TstT');
svmlwrite('Test',X1,Y1);
```

```
Runtime (without IO) in cpu-seconds: 0.00
Accuracy on test set: 96.00% (288 correct, 12 incorrect, 300 total)
Precision/recall on test set: 96.62%/95.33%

err_rate =

    0.0400

??? Error: File: PROJmain.m Line: 38 Column: 24
The expression to the left of the equals sign is not a valid target for an assignment.

Writing  100 200 300 400 500 600 done.
Writing  100 200 300 done.

Calling SVMlight:
svm_learn -c 100 -t 0 Train model

Scanning examples...done
Reading examples into memory...100..200..300..400..500..600..OK. (600 examples read)
Optimizing.............................................................................................
Optimization finished (0 misclassified, maxdiff=0.00099).
Runtime in cpu-seconds: 0.21
Number of SV: 88 (including 0 at upper bound)
L1 loss: loss=0.00000
Norm of weight vector: |w|=0.00907
Norm of longest example vector: |x|=3499.11360
Estimated VCdim of classifier: VCdim<=1008.17259
Computing XiAlpha-estimates...done
Runtime for XiAlpha-estimates in cpu-seconds: 0.00
XiAlpha-estimate of the error: error<=13.67% (rho=1.00,depth=0)
XiAlpha-estimate of the recall: recall=>85.00% (rho=1.00,depth=0)
XiAlpha-estimate of the precision: precision=>87.33% (rho=1.00,depth=0)
Number of kernel evaluations: 29199
Writing model file...done

Calling SVMlight:
svm_classify Test model predictions

Reading model...OK. (88 support vectors read)
Classifying test examples..100..200..300..done
Runtime (without IO) in cpu-seconds: 0.00
Accuracy on test set: 96.00% (288 correct, 12 incorrect, 300 total)
Precision/recall on test set: 96.62%/95.33%

accuracy =

    0.9600

>>
```

Figure 1.3: *The main MATLAB screen with the Workspace highlighted in red.*

1.2.4 The Current Directory setting

Once we start loading data, we have to give MATLAB an idea where to look for the external files we are interested in. If you place all you data files etc in a single location then you can change to that directory so that MATLAB will know where to find files with specific names.


MATLAB 7.6.0 (R2008a)
File Edit Debug Parallel Desktop Window Help
Current Directory /forrest/work/cs/SVM/Youn
Current Directory
Shortcuts How to Add What's New
Current Directory Workspace
Stack: Base
Command History
Command Window
New to MATLAB? Watch this Video, see Demos, or read Getting Started.
Start
OVR

Name	Value	Min	Max
C	100	100	100
X	<600x784 doubl...	0	255
X1	<300x784 doubl...	0	255
Y	<600x1 double>	-1	1
Y1	<300x1 double>	-1	1
accuracy	0.9600	0.96...	0.96...
ans	0	0	0
digit23Trn	<600x784 doubl...	0	255
digit23Tst	<300x784 doubl...	0	255
digit23TstT	<300x1 double>	-1	1
err	12	12	12
err_rate	0.0400	0.04...	0.04...
kernel	0	0	0
options	<1x1 struct>		
predictions	<300x1 double>	-1	1
sigma	0.5000	0.50...	0.50...
x	300	300	300

```
load predictions
predictions
predictions = sign(predictions)
%-- 4/17/08  9:42 PM --%
SA
loss
%-- 4/18/08 11:06 AM --%
svmclassify
%-- 4/19/08  3:33 PM --%
%-- 4/19/08  3:38 PM --%
X= load('digit23Trn.mat')
X = X.digiti23Trn
X = X.digit23Trn
Y = load('digit23TrnT')
X = load('digit23Trn.mat');
X = X.digit23Trn;
Y = load('digit23TrnT');
svmlwrite('Train',X,Y);
X1 = load('digit23Tst.mat');
X1 = X1.digit23Tst;
Y1 = load('digit23TstT');
svmlwrite('Test',X1,Y1);
```

```
Runtime (without IO) in cpu-seconds: 0.00
Accuracy on test set: 96.00% (288 correct, 12 incorrect, 300 total)
Precision/recall on test set: 96.62%/95.33%

err_rate =

    0.0400

??? Error: File: PROJmain.m Line: 38 Column: 24
The expression to the left of the equals sign is not a valid target for an assignment.

Writing  100 200 300 400 500 600 done.
Writing  100 200 300 done.

Calling SVMlight:
svm_learn -c 100 -t 0 Train model

Scanning examples...done
Reading examples into memory...100..200..300..400..500..600..OK. (600 examples read)
Optimizing..........................................................................................
Optimization finished (0 misclassified, maxdiff=0.00099).
Runtime in cpu-seconds: 0.21
Number of SV: 88 (including 0 at upper bound)
L1 loss: loss=0.00000
Norm of weight vector: |w|=0.00907
Norm of longest example vector: |x|=3499.11360
Estimated VCdim of classifier: VCdim<=1008.17259
Computing XiAlpha-estimates...done
Runtime for XiAlpha-estimates in cpu-seconds: 0.00
XiAlpha-estimate of the error: error<=13.67% (rho=1.00,depth=0)
XiAlpha-estimate of the recall: recall=>85.00% (rho=1.00,depth=0)
XiAlpha-estimate of the precision: precision=>87.33% (rho=1.00,depth=0)
Number of kernel evaluations: 29199
Writing model file...done

Calling SVMlight:
svm_classify Test model predictions

Reading model...OK. (88 support vectors read)
Classifying test examples..100..200..300..done
Runtime (without IO) in cpu-seconds: 0.00
Accuracy on test set: 96.00% (288 correct, 12 incorrect, 300 total)
Precision/recall on test set: 96.62%/95.33%

accuracy =

    0.9600

>> |
```

Figure 1.4: *The main MATLAB screen with the Current Directory highlighted in red.*

Chapter 2

Simple operations and working with variables

2.1 Using variables; scalars, vectors and matrices

When you start MATLAB you will see a symbol `>>`. This is the prompt where commands can be given to MATLAB. Lets start with something simple, using MATLAB like a pocket calculator. At the command prompt type:

```
>> 1+1
```

Once you press the *Enter* key, MATLAB will return the answer:

```
ans =

2
```

Of course we can do this with any combination of numbers and not just integers:

```
>> 24.7 + 2.1 + 5.6

ans =

32.4000
```

2.1.1 Simple scalar arithmetic

For those of you not familiar with the mathematical jargon, a *scalar* is simply a single number. In MATLAB a number of operations can be used to perform arithmetic with scalars.

Operation	Symbol	Example
Addition, a + b	+	3 + 22
Subtraction, a - b	-	90 - 54
Multiplication, a × b	*	3.14 * 0.85
Division, a ÷ b	/	56 / 8
Exponentiation, a^b	^	2 ^ 8

2.1.2 Order of evaluation

In MATLAB there is a so-called *order of evaluation* for different arithmetic operators. This means that the sequence in which MATLAB performs the different parts of a mathematical expression is fixed and we have to construct commands according to the following basic rules.

1. Parentheses (brackets), innermost first.
2. Exponentiation, left to right.
3. Multiplication and division with equal importance, left to right.
4. Addition and subtraction with equal important, left to right.

The best way to demonstrate the order of evaluation is using some basic examples. Lets take the mean of five different numbers:

3 6 4 10 7

Of course, we need to sum the numbers and divide by 5, we could do this as two separate steps:

```
>> 3 + 6 + 4 + 10 + 7

ans =

30

>> 30 / 5

ans =

6
```

We find the mean is 6, but from a practical point of view it is more convenient if we combine the steps of the calculations together. This is where the order of evaluation is important because we must make sure that the total of the numbers is found before the division step. For example, if we ignore the order of evaluation:

```
>> 3 + 6 + 4 + 10 + 7 / 5

ans =
   24.4000
```

In this case we get the wrong answer because the division was performed before the addition (item 3. of the order of evaluation comes before point 4.). In order to do the calculation properly we must employ parentheses:

```
   >> (3 + 6 + 4 + 10 + 7) / 5

   ans =

   6
```

From the order of operation we can see that the expression in parenthesis (item 1.) will be evaluated before the division (item 3.), so we have the correct sequence for the calculation to be evaluated properly.
Now we'll look at a less obvious example, which demonstrates that when two operations occur at the same point on the order of evaluation, the one on the left will be performed first.

```
   >> 3 / 4 * 5

   ans =

   3.7500

   >> (3 / 4) * 5

   ans =

   3.7500

   >> 3 / (4 * 5)

   ans =

   0.1500
```

The third result is different from the first two because the parenthesis force the multiplication to be evaluated before the division step even through it is on the right-hand side of the expression.
Here's a final example which employs exponentiation, we will calculate the value of 3^9:

```
   >> 3 ^ 9

   ans =

   19683
```

However, if we write:

```
>> 4 * 2 + 1

ans =

9

>> 3 ^ 4 * 2 + 1

ans =

163
```

We obtain the wrong answer because we have ignored the order of evaluation, to correct this statement we must use parenthesis to ensure that the `4 * 2 + 1` part of the expression is evaluated before the exponentiation is performed:

```
>> 3 ^ (4 * 2 + 1)

ans =

19683
```

Exercises: order of evaluation

Given that $2^{6.5} = 90.5097$ can you added parenthesis to the following MATLAB expression to obtain a value of 90.5097.

```
>> 12 / 4 - 1 ^ 4 - 2 + 3 * 1.5
```

Next can you insert only two pairs of parentheses into the following expression to again obtain a value of 90.5097.

```
>> 4 ^ 0.5 ^ 10 - 2 ^ 1 + 1 + 0.5
```

2.1.3 Using named variables

Imagine you are sent to the geological hardware store to buy equipment for a fieldwork trip. On your shopping list you have the following items you wish to buy:

- 6 Geological Hammers, cost 45.32 each.

- 2 Compasses, cost 23.17 each.
- 9 Pocket Magnifiers, cost 4.99 each.

To find the total cost of the shopping we can use MATLAB just like a pocket calculator, with the * symbol representing multiplication:

```
>> 6*45.32 + 2*23.17 + 9*4.99

ans =

363.1700
```

An alternative approach would be to store the number of each item in an assigned *variable*:

```
>> hammers = 6

hammers =

6

>> compasses = 2

compasses =

2

>> magnifiers = 9

magnifiers =

9
```

If you look at the *Workspace* window you will see that these assigned variables are listed in MATLAB's memory and we can use them to repeat the calculation:

```
>> hammers*45.32 + compasses*23.17 + magnifiers*4.99

ans =

363.1700
```

As a last step we can also assign a variable named `cost` to the result of our calculation:

```
>> cost = hammers*45.32 + compasses*23.17 + magnifiers*4.99

cost =
363.1700
```

If we wanted to find the average cost of the items on the shopping list, we also need to know the total number of items, we can perform this calculation and assign the result into a variable called `items`:

```
>> items=hammers + compasses + magnifiers

items =

17
```

Remember, the variable `cost` is still held in MATLAB's memory, so to find the average cost we just perform a division using the / symbol:

```
>> average_cost = cost / items

average_cost =

21.3629
```

When outputting variables it is sometimes useful to use the `;` symbol at the end of your expression. This means MATLAB will perform the calculation and store the result in the memory but it will not show the result on the screen. Use of the `;` symbol is essential when you dealing with large amounts of data and you don't want to slow down your work waiting for the screen to list through all the results. For example:

```
>> average_cost = cost / items

average_cost =

21.3629
```

Compared to:

```
>> average_cost = cost / items; %do not show the output
```

MATLAB won't accept variable names which are more than one word, therefore if we try to use `average cost` rather than `average_cost` we receive an error:

```
>> average cost = cost / items
???  Undefined function or method 'average'for input arguments of type 'char'.
```

At anytime you can see what variables are held in the MATLAB Workspace using the `whos` command:

```
   >> whos
```

```
Name         Size       Bytes      Class      Attributes
ans          1x1        8          double
average_cost 1x1        8          double
compasses    1x1        8          double
cost         1x1        8          double
hammers      1x1        8          double
items        1x1        8          double
magnifiers   1x1        8          double
```

In turn if you want to see the information stored in a variable, type its name at the command prompt and press Enter, for example:

```
>> hammers

hammers =

6
```

2.1.4 Rules for naming variables

As we saw in the previous example, the names assigned to variables have to follow some basic rules. In MATLAB the rules for variable names are as follows:

- Variable names must start with a letter.
- Names can only be based on the letters A-Z & a-z, numbers 0-9 and underscore _.
- Only the first 31 characters in a variable name are recognized by MATLAB
- Names are case sensitive: `name`, `Name`, `NAME` are different.

2.1.5 Navigating the Command History and updating variables

In section 2.1.3 we created some variables and worked with basic commands. One of the useful facilities provided by MATLAB is a *Command History*, which stores your previous commands in the memory. The commands are recorded in full, in the order of their execution, in the *Command History* window, where they can be selected using the mouse and reused by clicking the right mouse button and choosing the *Evaluate Selection* option.

The Command History is also available from the command prompt, pressing the up cursor key will cycle through your previous commands. After a long MATLAB session you may wish to recall a much earlier command quickly, this can be done by typing the beginning of the command and using the up cursor key. In this case MATLAB will only cycle through the previous commands which started with the same beginning.

Lets return to the example of shopping at the geological hardware store, which we used in section 2.1.3. We have now decided that we would like to buy 7 geological hammers rather than 6, to update our calculations we must first change the values of the variable `hammer`. We can recall the command we used to set the variable simply by typing the first few letters and pressing the up cursor key:

>> `ham` Now press the *up* cursor

```
>> hammers*45.32 + compasses*23.17 + magnifiers*4.99
```

This is the last command to begin with `ham`, but it's not the one we want so we press the *up* cursor again.

```
>> hammers = 6
```

This is the command we want, simply modify it to the new value and press enter.

```
>> hammers = 7

  hammers =

  7
```

It is important to realize that although we have changed the value of `hammers`, the other variables which depend on it, for example `cost` and `items`, **have not** been updated to take the new value into consideration. Therefore to update the other variables we have to repeat the calculations we performed earlier. This is where the *Command History* is useful, because we don't need to type the commands again, we can simply recall them.

```
>> cost = hammers*45.32 + compasses*23.17 + magnifiers*4.99

cost =

408.4900

>> items=hammers + compasses + magnifiers

items =

18

>> average_cost = cost / items

average_cost =

22.6939
```

2.1.6 Simple Arrays

In many cases we would like to use variables that contain more than a single number, these are arrays. Arrays can be rows or columns of numbers (called vectors) or higher dimensional matrices (two-dimensions or higher). To form an array in MATLAB you should use the square bracket symbols [and]. First lets make a *row* vector which contains the numbers 12, 35.2, and 65.

```
>> x = [12 , 35.2 , 65]

x =

12.0000 35.2000 65.0000
```

The `,` symbol places the numbers in a row. To place the numbers in a column use the `;` symbol:

```
>> x = [12 ; 35.2 ; 65]

x =

12.0000
35.2000
65.0000
```

There are a number of shortcuts when making vectors, for example, we can produce a row vector which contains numbers starting at 0.8 and finishing at 1.4 spaced at 0.15 intervals:

```
>> x = [0.8 :  0.15 :  1.4]

x =

0.8000 0.9500 1.1000 1.2500 1.4000
```

Note the use of the `:` operator to tell MATLAB to make an equal interval array. To make a column with the same numbers we have to add the transpose command `'` after the square brackets:

```
>> x = [0.8 :  0.15 :  1.4]'

x =

0.8000
0.9500
1.1000
1.2500
1.4000
```

If we have a long vector we may only be interested in certain parts of it, we can access those parts using an index number. Index numbers represent specific positions within an array starting from the first position which is given an index of 1. For example if we want to know the value of `x` at a given position:

```
>> x = [0.8 :  0.15 :  1.4];
>> x(3) %return the 3rd element of x

ans =

1.1000

>> x(5) %return the 5th element of x

ans =

1.4000

>> x(1:3) %the 1st through 3rd elements of x

ans =

0.8000 0.9500 1.1000

>> x([1 , 4 , 3]) %return the 1st, 4th and 3rd elements of x

ans =

0.8000 1.2500 1.1000

>> x([1 :  2 :  5]) %return the 1st, 3rd and 5th elements of x

ans =

0.8000 1.1000 1.4000
```

Currently the variable `x` holds five elements, so if we ask for the sixth we will get an error because we are looking for a position which does not exist:

```
>> x(6) %return the 6th element of x
???  Index exceeds matrix dimensions.
```

You can also perform arithmetic between vectors and scalars. Construct a column vector, `x`, and then perform some simple calculations with it and a scalar, producing an output row vector `y`:

```
>> x = [0.8 :  0.15 :  1.4]';

>> y = x + 2 %Addition of 2 to all the elements in x

y =

2.8000
2.9500
3.1000
3.2500
3.4000

>> y = x - 3 %Subtraction of 3 from all the elements in x

y =

-2.2000
-2.0500
-1.9000
-1.7500
-1.6000

>> y = x * 1.2 %Multiply all elements by 1.2

y =

0.9600
1.1400
1.3200
1.5000
1.6800

>> y = x / 2.5 %Divide all the elements by 2.5

y =

0.3200
0.3800
0.4400
0.5000
0.5600

>> y = x(1:2) + 5 %Add five to the first two elements

y =

5.8000
```

```
5.9500
```

The last type of array we will consider is the matrix which is a two or high dimensional array. The two dimensional matrix contains rows and columns. To construct a matrix we can use the , (row command) and ; (column command) operators. Note the use of the ; symbol to start a new row.

```
>> x = [1 , 2 , 3 , 4 ; 5 , 6 , 7 , 8]

x =

1 2 3 4
5 6 7 8
```

When making a matrix all the rows must have the same number of columns (as above where each row has 4 elements). If you try to make a matrix with rows of different lengths an error will be returned.

```
>> x = [1 , 2 , 3 , 4 ; 5 , 6 , 7 , 8 , 9]
???  Error using ==> vertcat
CAT arguments dimensions are not consistent.
```

A matrix has both rows and columns, so to select elements we must use both row indices and column indices. The row index is always specified first, then the column index.

```
>> x = [5.4 , 7.1 , 3.3 , 2.8 ; 1.4 , 6.9 , 9.2 , 3.4]

x =

5.4000 7.1000 3.3000 2.8000
1.4000 6.9000 9.2000 3.4000

>> x(2,1) %return the element in the 2nd row and 1st column

ans =

1.4000

>> x(1,3) %return the element in the 1st row and 2nd column

ans =

3.3000

>> x(2,:)  %return all the elements in the 2nd row

ans =
1.4000 6.9000 9.2000 3.4000

>> x(:,4) %return all the elements in the 4th column

ans =

2.8000
3.4000
```

2.1.7 Array-Scalar and Array-Array Arithmetic

When performing addition and subtraction on arrays you must either use an array and a scalar together or two arrays of the same size. As a first example we will create a matrix and add a scalar to each of its entries.

```
>> x = [5.4 , 7.1 , 3.3 , 2.8 ; 1.4 , 6.9 , 9.2 , 3.4]

x =

5.4000 7.1000 3.3000 2.8000
1.4000 6.9000 9.2000 3.4000

>> x + 1 %add 1 to each entry in x

ans =

6.4000 8.1000 4.3000 3.8000
2.4000 7.9000 10.2000 4.4000
```

Addition of two arrays of the same size means they are added together element by element:

```
>> x + [1 , 2 , 3 , 4 ; 5 , 6 , 7 , 8]

ans =

6.4000 9.1000 6.3000 6.8000
6.4000 12.9000 16.2000 11.4000
```

Of course, subtraction will work in the same way. Addition or subtraction of arrays with different sizes will cause an error:

```
>> [1 , 2 , 3 ] + [5 , 6 , 7 , 8]
 ???  Error using ==> plus
Matrix dimensions must agree.
```

There are two types of multiplication and division; *element by element* and *matrix*. With arrays of the same size *element by element* multiplication, division and exponentiation uses the `.` operator. We'll form two matrices and then perform some *element by element* operations.

```
>> x = [1 , 1 , 1 ; 2 , 2 , 2]

x =

1 1 1
2 2 2

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

y =

1 2 3
4 5 6

>> x.*y %element by element multiplication

ans =

1 2 3
8 10 12

>> x./y %element by element division

ans =

1.0000 0.5000 0.3333
0.5000 0.4000 0.3333

>> x.^y %element by element exponentiation

ans =

1 1 1
16 32 64
```

If you don't use the `.` operator MATLAB will perform matrix multiplication or division, but the arrays must be of the correct size, for example if we try to multiply the arrays `x` and `y`:

```
>> x*y %matrix multiplication of x and y
???  Error using ==> mtimes
Inner matrix dimensions must agree.
```

```
>> x = [1 , 1 , 1 ; 2 , 2 , 2]'

x =

1 2
1 2
1 2

>> x * y %matrix multiplication of x and y

ans =

9 12 15
9 12 15
9 12 15
```

Chapter 3

MATLAB Functions

3.1 Script M-files and the MATLAB Editor

As you can probably see already, when working with commands in MATLAB you will often want to repeat and change specific things as you develop your code. The best way to do this is using the built-in M-Editor to work with a sequence of commands which you can execute at any time. The commands can be stored in an M-file (with the extension `*.m`), so you can keep a record of your work and gradually develop the code. To create a new M-file:

1. Click on *File* in the main MATLAB window
2. Select *New*
3. Select *M-file*

the editor will appear with a blank document. We can now store our commands here (they must be kept in the correct sequence) and execute them at anytime by giving the name of the M-file in the command window, or highlighting specific commands with the mouse, giving a right-click and choosing "evaluate selection". I have made an example M-file with the commands we used in the previous example, it has the name `shopping.m` and can be loaded from the command window by typing:

```
>> open shopping
```

Using the editor provides us with a much more flexible way to issue commands to MATLAB, we can review the overall structure of a piece of code and gradually make modifications and improvements. Within the editor we can also add comments to the M-code, these are notes which MATLAB will ignore but provide a description of the code to remind us what is happening. Once we place a `%` symbol in the code, MATLAB will ignore the remainder of the line. The editor highlights comments in green, you can see an example of comments in the `shopping.m` file.

Listing of shopping.m:

```
hammers = 6 %define number of hammers
```

```
compasses = 2 %define number of compasses
magnifiers = 9 %define number of magnifiers

%in the next line sum the total cost
cost = hammers*45.32 + compasses*23.17 + magnifiers*4.99
items = hammers + compasses + magnifiers %number of items
average_cost = cost /items %calculate average cost
```

3.2 Working with “inbuilt” functions

The real power of MATLAB comes from its thousands of inbuilt functions. Each function is a sequence of MATLAB commands which will perform a specific task. Each function has a specific set of input and outputs according to the information it requires and the result it will produce. Later we will look at how we can write our own functions to perform a given task, but for the time being we will look at inbuilt functions to understand how they work.

3.2.1 Taking the square root

To find the square root of a number is simple and we could use standard MATLAB arithmetic commands. For example to find the square root of 9:

```
>> 9^0.5

ans =

3
```

We can perform the same task on a vector as long as we use the element by element approach:

```
>> [9 , 16 , 25].^0.5 %note the use of the dot

ans =

3 4 5
```

This is easy enough, but MATLAB also contains an inbuilt function called *sqrt* which can perform this task for us. To use a function we give its name at the command line and place the input within brackets. The input can be a series of numbers, a variable or an expression including other variables, finally the output of the function can also be assigned a specific variable name.

```
>> sqrt(9)

ans =
3
```

The `sqrt` function can also be used for vectors, matrices and expressions:

```
>> x = [9 , 16 , 25]

x =

9 16 25

>> sqrt(x)

ans =

3 4 5


>> y = sqrt(2*x+4) %assign the output to y

y =

4.6904 6.0000 7.3485
```

3.2.2 The `help` command

If you want information on what a function does and what its inputs and outputs should be then you can used MATLAB's help system. Simply give the **`help`** command followed by the name of the function you are interested in. This will provide detailed information on what the function does, other related functions which might also be useful, and a link to the function's page in the help documentation.

```
>> help sqrt

SQRT(X) is the square root of the elements of X. Complex
results are produced if X is not positive.

   See also sqrtm, realsqrt, hypot.

   Overloaded functions or methods (ones with the same name in other directories)
   help sdpvar/sqrt.m
   help ncvar/sqrt.m
   help sym/sqrt.m

   Reference page in Help browser
```

```
doc sqrt
```

3.2.3 Finding the right function

As I mentioned above, MATLAB has thousands of inbuilt functions and sometimes it can be difficult to guess the name of the function you need to perform a certain task. In these situations the `lookfor` function is useful, because it will search for specific keywords within all the available functions and return a list of matches. For example, if we want to take the square root of a number but we don't know the name of the function to use, we can use the `lookfor` function to get a list of possibilities:

```
>> lookfor('square root')

HYPOT Robust computation of the square root of the sum of squares
REALSQRT Real square root.
SQRT Square root.
SQRTM Matrix square root.
ipexConformalForward1 Forward transformation with positive square root.
ipexConformalForward2 Forward transformation with negative square root.
SQRT Symbolic matrix element-wise square root.
```

Lets take a slightly more complicated example. We want to generate a matrix with 3 rows and 2 columns that contains random numbers taken from a normal distribution. Our first step is to search for a function which can perform this task:

```
>> lookfor('random numbers')

RAND Uniformly distributed pseudo-random numbers.
RANDN Normally distributed random numbers.
RANDG Gamma random numbers (unit scale).
```

From the list it looks like the function `randn` will provide us with random numbers from a normal distribution. Next we'll look at its help entry to get more details concerning the inputs we should use:

```
>> help randn

RANDN Normally distributed random numbers.
R = RANDN(N) returns an N-by-N matrix containing pseudo-random values
drawn from a normal distribution with mean zero and standard deviation
one.  RANDN(M,N) or RANDN([M,N]) returns an M-by-N matrix.  RANDN(M,N,P,...)
or RANDN([M,N,P,...])  returns an M-by-N-by-P-by-...  array.  RANDN with
no arguments returns a scalar.  RANDN(SIZE(A)) returns an array the
same size as A.
```

This tells us the the function will return a matrix of normally distributed random numbers, the first input to the function is the number of rows for the matrix and the second is the number of columns (remember we wanted 3 rows and 2 columns). We'll name our output from the function `z`.

```
>> z = randn(3,2)

z =

-0.4326 1.1909
-1.6656 -1.1465
0.1253 0.2877
```

3.2.4 Multiple inputs and outputs

It is also possible for functions to work with multiple inputs and outputs. A good example is the `sortrows` function which sorts a matrix of values from lowest to highest based on the values in one of its columns. The primary output of the function is the sorted matrix, but a secondary output giving the index of the sorted values is also available. First we will sort the random numbers in `z` according to the values in the first column (if we don't specify which column to base the sort upon, MATLAB will automatically use the first).

```
>> y = sortrows(z) %sort the rows according to the first column

y =

-1.6656 -1.1465
-0.4326 1.1909
0.1253 0.2877
```

Now we'll also use the second (optional) output of the function that records the original indices of the sorted values (which we will call `idx`). Note that in order to obtain multiple outputs we have to group them together in square brackets when calling the function:

```
>> [y,idx] = sortrows(z) %sort the rows by the first column and return the indices

y =

-1.6656 -1.1465
-0.4326 1.1909
0.1253 0.2877


idx =
2
1
3
```

Finally, we'll supply the function with a second (optional input argument) which gives the index of the column upon which the sort should be based. In this case we will sort according to the values in the second column:

```
>> [y,idx] = sortrows(z,2)

y =

-1.6656 -1.1465
0.1253 0.2877
-0.4326 1.1909

idx =

2
3
1
```

3.3 Exercise: Mmmm, π

Now it's your turn. Throughout the class we will attempt to solve problems using MATLAB by applying the topics which have been discussed. Some of you may find them relatively easily (especially if you have some experience of MATLAB) and others will find them more difficult. I have specifically designed the exercises to be challenging, so don't expect to finish them within a few minutes. It is important to realize that MATLAB is a tool and will only help you to solve a problem if you know how the problem should be solved. The first question you should ask yourself is "what method will I used to solve the problem?" and only then should you think about "how can I implement my method in MATLAB?". For this first exercise we are going to look at a simple numerical problem involving π.

Approximating π

The value of π has a very simple definition, "the ratio of a circle's circumference to its diameter", but it is irrational number, meaning its decimal representation never ends or repeats.

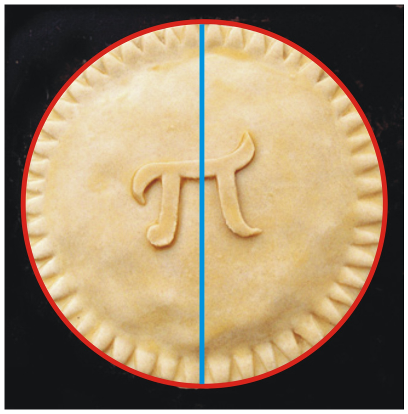


Figure 3.1: *The π pie. If the pie is a perfect circle then the ratio of its circumference (red line) to its diameter (blue line) is equal to π*

.

Determining a more and more accurate evaluation of π is still an active area of research, current estimates have a precision of over 100000000000 decimal places (at the beginning of 2010 Fabrice Bellard announced that he had calculated π to 2.7 trillion digits). To give you a flavor of what some people's research involves, the first 1000 digits of π are:

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862 80
348253421170679821480865132823066470938446095505822317253594081284811174502841027019385
211055596446229489549303819644288109756659334461284756482337867831652712019091456485669
234603486104543266482133936072602491412737245870066063155881748815209209628292540917153
643678925903600113305305488204665213841469519415116094330572703657595919530921861173819
326117931051185480744623799627495673518857527248912279381830119491298336733624406566430
860213949463952247371907021798609437027705392171762931767523846748184676694051320005681
271452635608277857713427577896091736371787214684409012249534301465495853710507922796892
589235420199561121290219608640344181598136297747713099605187072113499999983729780499510
597317328160963185950244594553469083026425223082533446850352619311881710100031378387528
865875332083814206171776691473035982534904287554687311595628638823537875937519577818 57
780532171226806613001927876611195909216420198 9

Over time people have used a number of different approximations for π, a popular one is $\frac{22}{7}$. Probably the strangest is a legal bill which was proposed in 1897 that would set $pi = 3.2$ in order that people could remember it more easily. There is a long history of methods with which to approximate π, one of the most elegant of which was given in the 14th century by the Indian mathematician Madhava of Sangamagrama (interestingly the technique was "rediscovered" in Europe over 300 years later). Madhava of Sangamagrama showed that $\frac{\pi}{4}$ could be obtained as the sum of an infinite series:

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \frac{\pi}{4}$$

Therefore if we work with the first 4 terms, for n=0 to n=3 and ultimately working to infinity we would get:

$$\frac{(-1)^0}{2*0+1}+\frac{(-1)^1}{2*1+1}+\frac{(-1)^2}{2*2+1}+\frac{(-1)^3}{2*3+1}+\dots\frac{(-1)^\infty}{2*\infty+1}=\frac{\pi}{4}$$

the beginning of which simplifies to:

$$1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\dots=\frac{\pi}{4}$$

We're going to use this approach to estimate π in MATLAB. Obviously we can't calculate an infinite number of terms so we'll only work to a maximum of $n = 10000$. One function you might find useful is `sum`, look at its help entry to get more information about its use. For those of you familiar with MATLAB this exercise won't be too much of a challenge so you should try to perform same task whilst playing MATLAB "golf". That is, what is the smallest number of characters you can use in your code to make the calculation (mine was 30 characters). I would recommend that you work in the editor in order that you can develop and modify your code more easily.

Steps to completing the exercise

There are three main tasks to solve this exercise:

1. Generating the numbers between 0 and 10000 to act as n.
2. Calculating the terms of the series using the values of n.
3. Finally, summing the series and multiplying by 4 to estimate π.

Functions you may find useful

- `sum` - Sum of array elements.

3.4 Writing your own functions

One of the great advantages of MATLAB is the capacity to write your own functions with specific inputs and outputs. This means you can design and write your own processing routines which can be used over and over again with no additional effort (always a good thing!).

3.4.1 Inline functions

If you have a simple expression with a single output you can define it as an inline function so that it can be used repeatedly without retyping the code. For example, imagine we want to study the equation for a variety of values f and θ:

$$g = \sin(2 * \pi * f + \theta)$$

We can use the `inline` function to define this expression and its input, so that it can be called repeatedly. Note that during the definition of the inline function the expression and input variable names are defined as character strings.

```
>> g = inline('sin(2*pi*f + theta)', 'f', 'theta')

g =

Inline function:
g(f,theta) = sin(2*pi*f + theta)

>> g(0.5,pi./4)

ans =

-0.707106781186547
```

Inline functions are useful for simple expressions, but in more complex situations it is necessary to create an M-function.

3.4.2 M-functions

It is simple to write your own M-functions with specific input and outputs. An M-function is a separate M file (normally written in the MATLAB editor), which is a sequence of MATLAB commands, working with provided inputs and producing a specific output. The key to an M-function is the first line of the function, which defines 4 things in order:

1. That the file contains a function
2. The output variables of the function
3. The name of the function
4. The input variables of the function

We'll start with a example of a function called `simple`, which has one input `x` and one output `y`. The operation of the function takes the value of `x`, adds one to it and outputs the result as `y`. Start a new M-file in the editor and enter the following text.

```
function y=simple(x)

y=x+1;
```

When you try to save the function, MATLAB will suggest the same name as you defined for the function (`simple.m`). When searching for a function MATLAB actually refers to the filename rather than the defined name inside the function, so it's essential that they match each other. Now we can call the function `simple` in the same way as any other MATLAB function. For example:

```
>> x = 5
>> y = simple(x)
y =

6
```

The the names of the input, `x`, and output, `y`, are only used within the function, so when we are calling the function externally (i.e. from outside) we can use any names we want. For example:

```
>> a = 5
>> b = simple(a)
b =

6
```

One of the advantages of M-functions is that all the variables which may be created by the intermediate steps of the function will not be stored in the MATLAB memory once the calculation is complete, only the output is stored. This makes functions a very tidy way of performing certain tasks because only the output is stored. We can set multiple inputs and outputs simply by adding them to the first line of the function. The next function we'll write is called `simple2.m`, which takes two inputs `c` and `d` and gives two outputs, `e` which is the difference of `c` and `d`, and `f` which is the sum of `c` and `d`. Note that comments can also be included to explain the steps of the function.

```
function [e,f]=simple2(c,d)

e = c - d; %find the difference
f = c + d; %find the sum
```

Now we must supply two input names and two output names. For example:

```
>> cats=7;
>> dogs=4;
>> [shirt,socks]=simple2(cats,dogs)

shirt =

3
```

```
socks =

11
```

Hopefully this will convince you that you can use any variable names, of course it is always best to use meaningful names which describe the variable.

Including help and comments

One key aspect of writing functions is including detailed help so that other people can use your function and comments which show how the calculations are performed. Often you will write a function and after not looking at it for a few months you will have forgot how it works or how to use it. In these situations the help information and comments are invaluable. Help is entered after the first line of the file and is basically a series of comments, however, because of their position MATLAB can find them whenever you use the `help` or `lookfor` functions.
We take an example from probably the worlds most famous equation, the mass-equivalence relationship:

$$E = mc^2$$

where E is energy, m is mass and c is the speed of light. We can write a *vanilla* function (i.e. the most simple form) without help or comments.

```
function E=massequiv0(m)

c = 299792458;
E = m*c^2;
```

In this form the function will work and return the correct answer. It is obvious what is happening because we are only performing a simple operation. However some functions require thousands of lines of code, thus it is very easy to become lost and the meanings of different variable names maybe obscure. Therefore it is always a good idea to include some comments that describe what is happening in the different lines of code. For an equation such as this it is also a good idea to state which units are being used.

```
function E=massequiv1(m)

c = 299792458; %speed of light [m s^-1]
E = m.*c.^2; %calculate the energy in Joules [kg m^2 s^-2]
```

Things are starting to improve, but we still have to include the help information. The help is essential because it describes what the function does and what the different inputs and outputs represent.

```
function E=massequiv2(m)
%massequiv2 - Find energy from mass based on Einstein (1905)
%
% Syntax:  energy = massequiv2(mass)
%
% Inputs:
% m - mass [kg]
%
% Outputs:
% E - equivalent energy [Joules or kg m^2 s^-2]
%
% Other m-files required:  none
% Subfunctions:  none
% MAT-files required:  none
%
% Author:  Dave Heslop
% Department of Geosciences, University of Bremen
% email address:  dheslop@uni-bremen.de
% Last revision:  6-Dec-2008

%------------- BEGIN CODE --------------

c = 299792458; %speed of light [m s^-1]
E = m.*c.^2; %calculate the energy in Joules [kg m^2 s^-2]

%------------- END CODE --------------
```

With the help information in place we can use the `lookfor` and `help` functions:

```
>> lookfor Einstein
>> help massequiv2
```

3.5 Exercise: Mmmm, a second helping of π

In this exercise simply convert the script file you created for the evaluation of π into a M-function. The input for the function should be `N`, the number of terms to include in the evaluation and the output `p` is the final evaluation of π. Don't forget to include help and comments.

Steps to completing the exercise

There are three main tasks to solve this exercise:

1. Add a header line defining the function name (`madhava`), the input (`N`) and output (`p`).
2. Define the values of n to be used as ranging between `0` and `N`
3. Add suitable help and comments (see previous examples)

Functions you may find useful

- `sum` - Sum of array elements.

3.6 Exercise: The Geiger counter

Often a Geiger counter will be used to measure the radioactivity of rocks. One problem with simple Geiger counters is that at high activity (high count rates) they will underestimate the true activity. This is because after a γ-ray enters the detector the system is "dead" for a short period of time during which it cannot measure. If another γ-ray enters the system during this dead-time it will not be counted and thus the observed count rate will be less than the true count rate. This situation is made worse because not only are the γ-rays which enter during the dead-time not counted, they do extend the dead-time.

Figure 3.2: *How can we find the true count rate of a Geiger counter which suffers from dead-time.*

The relationship between the observed count rate (N_{obs}) and the true count rate (N_{true}) is an exponential law equation:

$$N_{obs} = N_{true} e^{-N_{true}\tau}$$

where τ is the dead-time per pulse (20 x 10^{-6} seconds in old instruments). Note this is a transcendental equation which means that it cannot be rewritten in the form $N_{true} =$ .... In this exercise you will write an M-function to determine the value of N_{true} for a given input value of N_{obs} and τ.

Steps to completing the exercise

When writing your M-function you will need to think about the inputs and outputs the function will require. The required inputs should follow the basic lines of:

- The observed count rate (called `Nobs`)
- The dead-time (called `tau`)
- The array of integer count rate values to test (called `Ntest`)

For the calculation itself we need to determine the calculated count rate N_{calc} for each of the guesses in `Ntest` and then find which one fits best with the observed count rate, N_{obs}. This will involve the following steps:

1. Determine the values of N_{calc} which would be measured for your values of N_{true}.
2. Determine which value of N_{calc} is closest to the value of N_{obs} measured on the Geiger counter.
3. Output the value of `Ntest` corresponding to the best prediction of N_{obs}.

Functions you may find useful

- `exp` - Exponential
- `sort` - Sort array elements in ascending or descending order.
- `sqrt` - Square root

Chapter 4

Relational and logical operators

4.1 Relational and logical operations

The purpose of relational and logical operators is to provide answers to True/False questions. To represent this idea numerically MATLAB uses a system based on:

- True = 1
- False = 0

4.1.1 Relational testing

As an example, relational operators can be used to test if two numbers are equal, or if one number is greater than another. MATLAB uses the following relational operators:

Relational Operator	**Description**
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
~=	not equal to

These operators can be used to compare both scalars and arrays. For example, working with scalars:

```
>> A = 5;
>> B = 6;

>> A == B %test if A is equal to B

ans =

0
```

```
>> A ~= B %test if A does not equal B

ans =

1

>> A >= B %test if A is greater than or equal to B

ans =

0

>> A < B %test if A is less than B

ans =

1
```

When we work with arrays the relationships are tested on a *element-by-element* basis. The output will be the same size as the array being tested, with the position of ones and zeros in the output corresponding to the elements being tested:

```
>> A = [1:1:5]

A =

1 2 3 4 5

>> B = [5:-1:1]

B =

5 4 3 2 1

>> A >= 4 %positions where A is greater-than or equal to 4

ans =

0 0 0 1 1

>> A == B %positions where A equals B

ans =

0 0 1 0 0

>> B < A %positions where B is less-than A
```

```
ans =

0 0 0 1 1
```

In the case where the two arrays are not the same size, MATLAB will return an error because it is not possible to perform an element-by-element comparison:

```
>> A = [1:1:4]

A =

1 2 3 4

>> B = [1:1:3]

B =

1 2 3

>> A == B
???  Error using ==> eq
Matrix dimensions must agree.
```

A particularly useful function which extends the use of the relational operators is `find`. This function returns the indices of positions within an array which return a `True` from the test relation. In the following example we'll use `find` to locate the numbers in the array which are greater than or equal to three:

```
>> A = [5:-1:1]

A =

5 4 3 2 1

>> idx = find(A>=3) %index of values less than or equal to 3

idx =

1 2 3

>> A(idx) %test the result by calling the indexed values

ans =

5 4 3
```

4.1.2 Logical Operators

Logical operators allow us to combine a series of relational expressions. In this way we can specify in detail a series of relations which a given number must meet or not meet.

Logical Operator	**Description**
&	AND
|	OR
~	NOT

As a first example we'll compare to vectors `A` and `B` to find where `A` is greater than `B`, with the added condition that the only results which are acceptable are those where `A` does not equal 5.

```
>> A = [1:1:5]

A =

1 2 3 4 5

>> B = [5:-1:1]

B =

5 4 3 2 1

>> A > B & A~=5 %find A is greater than B and A does not equal 5

ans =

0 0 0 1 0
```

We can see from this result that the fifth element of `A` meets the first requirement (>`B`), but not the second (~=`5`), so a result of `False` is returned. Only the fourth element of meets both requirements. The logical operators can also be included when using the `find` function. We'll work with the same variable `B`, and find the values which are less than or equal to 2 **or** greater than 4.

```
>> idx = find(B<=2 | B>4)

idx =

1 4 5

>> B(idx) %check the result

ans =
5 2 1
```

4.1.3 Controlling the flow of your code

Relational operators are very useful for deciding on specific routes your code should take. For example, depending on a specific property of a variable you may wish to perform different calculations. MATLAB has a number of inbuilt keywords which allow you to control the flow of your code. The simplest example is `if`, which provides a switch to determine if a certain operation should be performed. In the next example we will test if a number is greater than 6.5, if it is then we will subtract a value of 3.1 from it.

```
>> C = 7.1; %set our value to be processed
>> if C > 6.5 %relational operator
     C = C - 3.1; %when True perform the calculation
>> end %end of the code relating to the if-statement

>> C %display the result

C =

4.0000
```

You can see MATLAB highlights the keywords in blue. In this case the value of `C` returned `True` when compared to the relational operator so the code between the `if` statement and the `end` statement was executed. If `C` had been a value which would have returned a `False` then the code between the `if` statement and the `end` statement would have been ignored. For example:

```
>> C = 6.1; %set our value to be processed
>> if C > 6.5 %relational operator
     C = C - 3.1; %when True perform the calculation
>> end %end of the code relating to the if-statement

>> C %display the result

C =

6.1000
```

As above the conditions for the `if` statement can be constructed from as many relational operators linked by logical operators as you want. We can add to the `if` statement using the keyword `else`. This provides an alternative procedure which should be performed when the relational operator in the `if` statement is found to be `False`. We'll extend our example from above so that if the input value is less than or equal to 6.5 a value of 9.3 is added to it.

```
>> C = 6.1; %set our value to be processed
>> if C > 6.5 %relational operator
      C = C - 3.1; %when True perform the calculation
   else %enter the alternative for a False
      C = C + 9.3;
>> end
>> C %display the result

C =

15.4000
```

Our value of 6.1 returned `False` when tested against the relational operator at the `if` statement, so the code corresponding to the `else` statement was executed. We can even make these alternative pieces of code conditional using the `elseif` statement and a further relational operator. We'll modify the above example in order that 9.3 is only added to the value if the starting number is less than 2.4.

```
>> C = 6.1; %set our value to be processed
>> if C > 6.5 %relational operator
      C = C - 3.1; %when if is True perform the calculation
>> elseif C < 2.4 %enter the alternative condition
      C = C + 9.3; %when elseif is True perform the calculation
>> end
>> C %display the result

C =

6.1000
```

4.1.4 For Loops

There are a number of other ways of controlling the flow of your code, for example the `for` command which is very useful when we want to repeat the same operation a specific number of times. For loops are commonly used in other programming languages such as FORTRAN to repeat specific tasks, they are also possible in MATLAB but computationally expensive, so they should be avoided when possible. For loops depend on a variable which is set as a counter that will run for a specific number of cycles. For example, we'll use a counter `i` which will loop between the value of 1 and 5, for each cycle of the loop we will print the value of `i` on the screen.

```
>> for i=1:5
      i %print the value of i on the screen
>> end

i =
1

i =
2

i =
3

i =
4

i =
5
```

We can see that the loop repeats between the `for` and `end` statements for each value of `i`, we can set as many `i` values and in any sequence we want. For example:

```
>> for i=[3 7 -1]
      i %print the value of i on the screen
>> end

i =
3

i =
7

i =
-1
```

For a more complicated example we will return to our approximation of π. If you remember we employed the infinite series of Madhava of Sangamagrama who showed that $\frac{\pi}{4}$:

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \frac{\pi}{4}$$

Therefore if we work with the first 4 terms, for n=0 to n=3 and ultimately working to infinity we would get:

$$\frac{(-1)^0}{2*0+1} + \frac{(-1)^1}{2*1+1} + \frac{(-1)^2}{2*2+1} + \frac{(-1)^3}{2*3+1} + \ldots \frac{(-1)^\infty}{2*\infty+1} = \frac{\pi}{4}$$

We can easily perform this calculation within a `for` loop, which the value of `n` increasing from `0` to `10000`. For each iteration we will simple add the new term to the existing estimate of π.

```
>> p=0; %set our starting estimate to 0
>> for n=0:1e5 %loop between 0 and 10000
```

```
      pnew=((-1).^n./(2*n+1))*4; %find the newest term
      p=p+pnew; %add the new term to the estimate
>> end %end the loop
>> p =
3.1416
```

One particularly useful feature of loops is that you can use the counter variable as an index to place data into an output variable. In this next example we want store each term of the series into an output array.

```
>> pterm=zeros(1e5,1); %setup an output array with zeros
>> for n=0:1e5 %loop between 0 and 10000
      pterm(n+1)=((-1).^n./(2*n+1))*4; %note the index is n+1
>> end %end the loop
>> pterm %view the output on the screen
```

We now have each term stored in sequence in variable `pterm`. Note that because the index of the first position in a MATLAB array is `1`, but out first value of `n` is `0`, we always have to use an index of `n+1`.

4.2 Exercise 1: Classifying sediments

Often sediments are grouped according to their grain size. In this example we'll look at the grain size fractions corresponding to:

- Sand, particles with sizes greater than 1/16 μm and less than 2 μm
- Silt, particles with sizes greater than 1/256 μm and less than 1/16 μm
- Clay, particles with sizes less than 1/256 μm

Data for 100 samples is stored in the file *grainsize.mat*, which contains a 100 x 3 matrix of data called `gs`. The numbers in the matrix correspond to the percentage of sand (first column), silt (second column) and clay (third column), where each sample is represented by one row (therefore each row of the matrix adds to 100%). Use the relational and logical operators we discussed above to answer the following questions.

1. How many samples contain more than 50% sand?
2. How many samples contain more than 50% silt?
3. How many samples contain more than 30% silt or more than 25% clay?
4. How many samples contain more than 40% sand, less than 40% silt and more than 40% clay?

Steps to completing the exercise

This exercise is pretty simple but is relies on two things:

- Correctly selecting the column of the data matrix which corresponds to the grain size fraction you are interested in.
- Constructing relational and logical operators to test which samples fulfill the given criteria.

To determine the number of samples which fulfill the a criteria you can either:

1. Find the sum of the array returned by the logical operator (i.e. add up the `1`s and *0*s).
2. Use the `find` function to return the index positions of the samples which meet the criteria and then determine the total number of returned positions.

Functions you may find useful

- `sum` - Sum of array elements.
- `numel` - Number of elements in array.

4.3 Exercise 2: The Safe Cracker

We're going to write a function that simulates trying to find the code which will open a safe full of money. The safe has a three-digit combination which must be entered into the function and the output must say wether the safe stays locked or opens (you can represent this numerically, for example a returned `0` represents "locked", a `1` represents "open").

Figure 4.1: *The safe requires the correct 3 digit combination to be opened*
.

Our safe, however, has a slight design problem which will help us to find the combination. The safe will tell us which digits are wrong and which are correct, for example:

```
The first digit is incorrect.
The second digit is correct.
The third digit is incorrect.
```

You can send text messages to the MATLAB command window using the function `disp`. Write an M-file using logical operators which will simulate this safe.

Steps to completing the exercise

This exercise as a whole has a number of things you must consider beyond simply how to test if a series of numbers are the same or not:

- Handling input (the guess of the combination) and output (indicator if the safe opens or not).
- Testing the guessed combination against the true combination.
- Determining if the safe should open or not.

You can approach these different parts using the various topics we have studied in the past chapters:

1. Input and output can be handled by constructing an M-function.
2. Testing if the digits of the guessed combination and true combination are the same can be performed using relational operators.

Functions you may find useful

- `disp` - Display text or array.

As a simple example here is a script with which we can test if two numbers A and B are the same and output the result to the screen. You should be able to adapt this example for your own solution:

```
A=5
B=6

if A==B
disp('The numbers are the same')
else
disp('The numbers are different')
end
```

4.3.1 An even worse safe

Once you have written your function, modify it so that the guess does not need to be so accurate for the lock to open. Construct the function so that a guessed digit only has to be within ±1 of the true number in order for it to be accepted as correct.

Steps to completing the exercise

This exercise only requires some simple modifications to your previous M-function. It might be a good idea to save your modified function with a new name so you don't overwrite your previous work. For the previous exercise you tested if the guessed digits were equal to the true combination, now you need to check if the guess digit is within ± 1 of the true combination. In other words you must test if a digit is greater than or equal to the the true digit minus 1 and less than or equal to the true digit plus 1, so you'll need to use a logical operator to test both cases simultaneously.

Chapter 5

Input / Output and data management

How can we import and export data from the MATLAB workspace? This can be a surprisingly complicated question to answer and depending on what kind of data you are working with you may want to consider a number of different approaches. Of course the advantage of MATLAB is that once you have found the best way to import your data you can write a function which will perform the task automatically.

5.1 Import

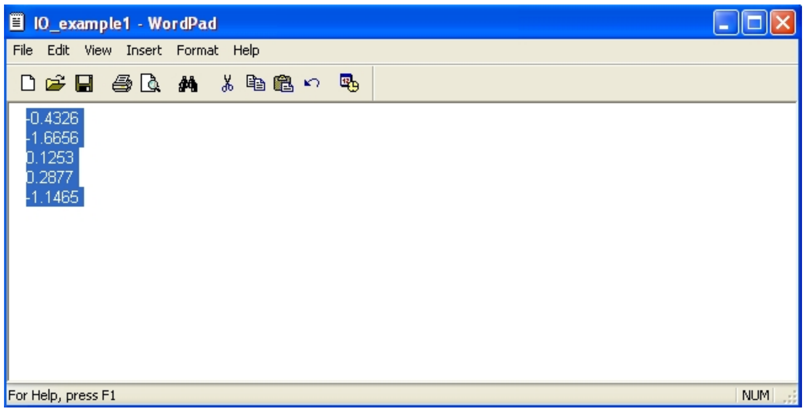
We'll look at the following ways in which you can import your data into MATLAB.

- Copy and Paste.
- Loading from an ASCII text file.
- Low-level I/O functions.
- Import from an EXCEL workbook.
- Reading from a NetCDF file.

Each of these methods has both advantages and disadvantages, so often it is best to give a little thought as to which technique will work best for your data. One important thing to remember is that MATLAB cannot mix text and numbers, so you can't have column headers as you would have in EXCEL.

5.1.1 Copy and Paste

This is probably the simplest import method but it is also the most cumbersome. We'll use the function `clipboard` (which requires the Sun Microsystems Java software) to paste copied data into MATLAB. Open the file **IO_example1.txt** in a text editor such as Notepad or Wordpad, select the data and copy it.



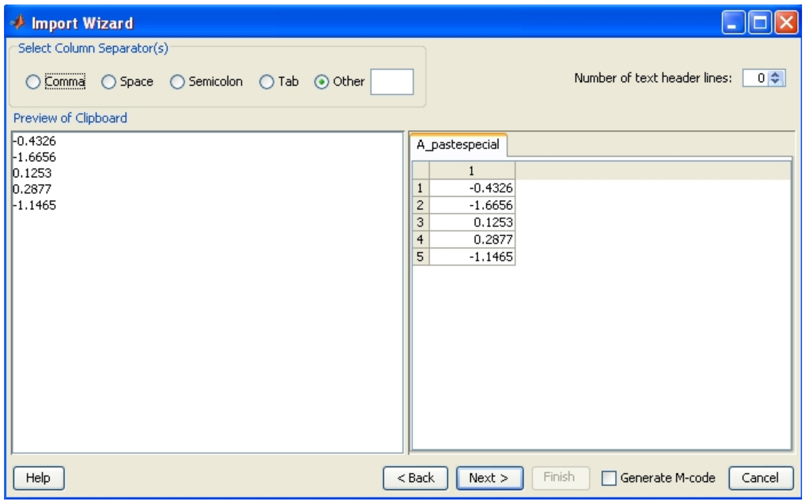
```
-0.4326
-1.6656
0.1253
0.2877
-1.1465
```

Figure 5.1: *The example data in text file IO_example1.txt.*

We can now paste the data from the clipboard with the command:

```
>> data = clipboard('pastespecial')
```

The import wizard will appear that allows you to check that your data looks okay. If you are satisfied then click `Next>` and then `Finish`.



Select Column Separator(s): Comma Space Semicolon Tab [✓ Other]

Number of text header lines: 0

Preview of Clipboard

```
-0.4326
-1.6656
0.1253
0.2877
-1.1465
```

A_pastespecial

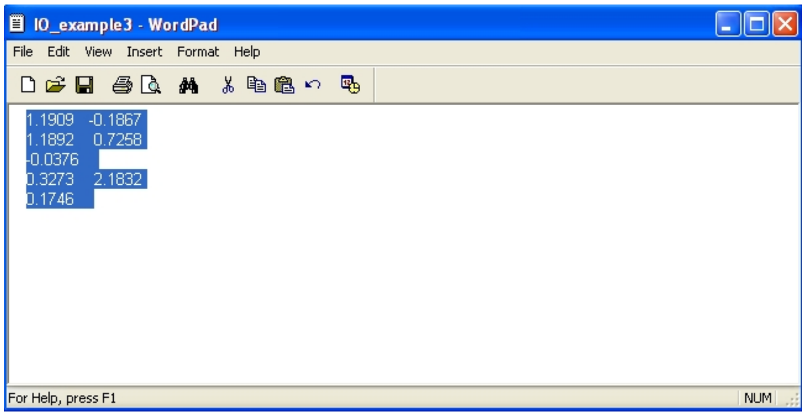
	1
1	-0.4326
2	-1.6656
3	0.1253
4	0.2877
5	-1.1465

Figure 5.2: *MATLAB's import wizard allows us to review the copied data*

This has created a structural array (an advanced type of data array which we won't be discussing in detail) called `data`. To convert `data` into a standard array we give the command:

```
>> data = data.A_pastespecial;
```

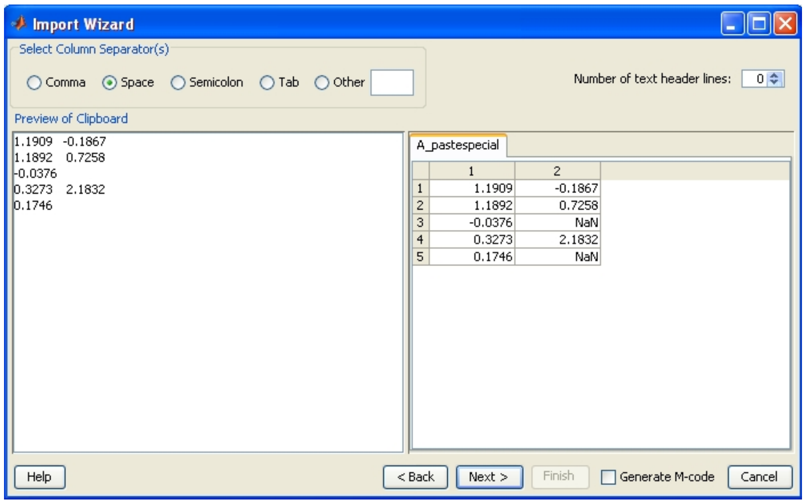
We can use the same procedure for matrices, select all the data in the file **IO_example2.txt** and repeat the import procedure. MATLAB can only import blocks of data which meet the requirements of scalars, vectors or matrices. In the file **IO_example3.txt** we can see that not all of the rows contain the same number of values (this could be a common situation when dealing with experimental data).



```
1.1909  -0.1867
1.1892   0.7258
-0.0376
0.3273   2.1832
0.1746
```

Figure 5.3: *The example data in text file IO_example3.txt.*

MATLAB automatically fills any gaps with `NaN` which stands for *Not a Number.*



Comma [✓ Space] Semicolon Tab Other

Number of text header lines: 0

```
1.1909  -0.1867
1.1892   0.7258
-0.0376
0.3273   2.1832
0.1746
```

	1	2
1	1.1909	-0.1867
2	1.1892	0.7258
3	-0.0376	NaN
4	0.3273	2.1832
5	0.1746	NaN

Figure 5.4: *The import wizard fills blank spaces with NaN symbols*

5.1.2 The `load` function

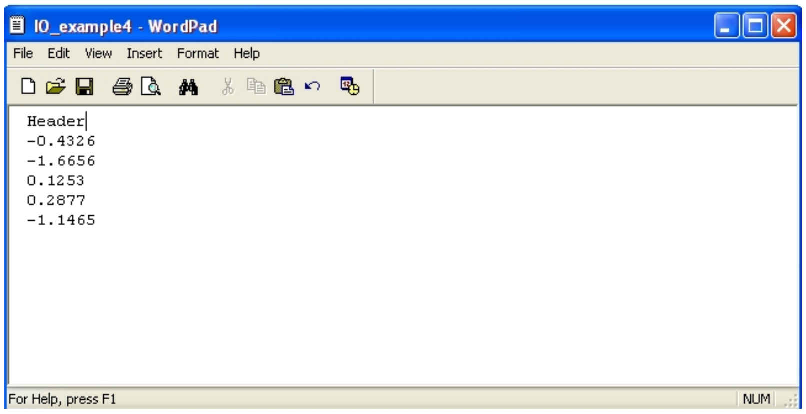
If your data is contained in a simple ASCII text file which does not contain headers etc, then you can read it straight into MATLAB using the `load` function, which requires a text string containing the name of the file to be imported.

```
>> data=load('IO_example1.txt')

data =

-0.4326
-1.6656
0.1253
0.2877
-1.1465
```

Unfortunately the `load` function cannot handle missing data values, so when we try to load **IO_example3.txt** which contained gaps in the second column, MATLAB will return an error.

```
>> data=load('IO_example3.txt')
??? Error using ==> load
Number of columns on line 2 of ASCII file F:\GLOMAR_MATLAB2009\IO\IO_example3.txt
must be the same as previous lines.
```

We also have a problem with header lines, for example, the file **IO_example4.txt** contains some text:



```
Header
-0.4326
-1.6656
0.1253
0.2877
-1.1465
```

Figure 5.5: *An example data file with a text header.*

MATLAB cannot combine text and numbers into one array so it returns an error when we try to load the file:

```
>> data=load('IO_example4.txt')
??? Error using ==> load
Unknown text on line number 1 of ASCII file F:\GLOMAR_MATLAB2009\IO\IO_example4.txt
"Header".
```

This means you have to be careful when preparing input files to make sure that they are in a form which MATLAB can understand. There are extensions to the `load` function which allow you to skip header lines. The function `textread` allows a number of optional input arguments which allow you to ignore problematic objects like header lines. As a very basic example we'll provide `textread` with a filename, tell it to read 64bit floating point numbers (`'%n'`) and ignore one header line at the top of the file:

```
>> data=textread('IO_example4.txt','%n','headerlines',1)

data =
-0.4326
-1.6656
0.1253
```

```
0.2877
-1.1465
```

There are a number of different options for `textread` which allow you, for example, to define delimiting characters and replacements for any missing values. Check the MATLAB help for more details on `textread` and the more advanced function `textscan`.

5.1.3 Low-level I/O functions

Often you will have complex data files which cannot be handled with the simple `load` or `textread` functions. For such situations MATLAB has a collection of inbuilt functions which assist you with the various input / output tasks. The functions are very useful if you are writing your own code to import specific types of data files. Here we will only look at a very basic example, but the functions can be combined together and used to make very complex input routines. In the following example we will open a text file, ignore a header line and read data into MATLAB line by line.

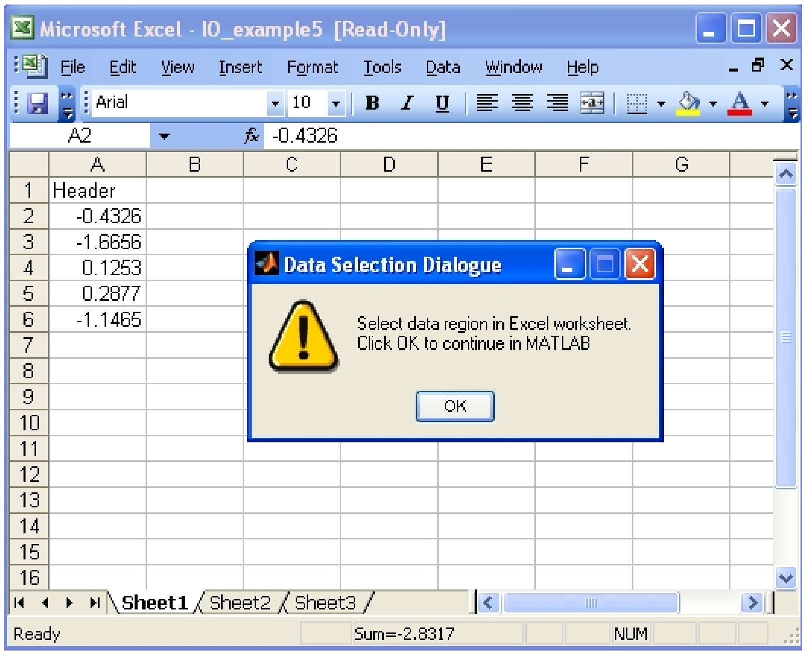
```
>> fname='IO_example4.txt' %define the file to be opened
>> fid=fopen(fname) %open the file and assign it an identification number

fid =

3

>> fgetl(fid) %read one line from the file (the header)
ans =
Header

>> input=fgetl(fid) %read the next line from the file
input =

-0.4326

>> x(1)=str2num(input) %convert the string into a number and store in x

x =

-0.4326

>> input=fgetl(fid) %read the next line from the file

input =

-1.6656

>> x(2)=str2num(input) %convert the string into a number and store in x
x =
-0.4326 -1.6656

>> fclose(fid) %close the file
```

Using these low level I/O functions provides the most flexible method to import and export data in MATLAB, but you will need some experience to be able to use the efficiently. You can look on the MathWorks File-exchange for examples of how to use the functions and there are also a number of freely-available functions which can help with import and export.

5.1.4 Import from an EXCEL workbook

For a number of years MATLAB has included a function `xlsread`, which (in theory) should allow you to import data directly from your existing MATLAB workbooks. Here we will look at a few examples of how the function can be used, but again it is important for you to check the available help information to find the options which will work best for your data. As a first step we will work interactively with a worksheet, we'll call the `xlsread` function, which will open the EXCEL workbook and allow us to select the data which should be imported. *N.B. if you're working with a German version of EXCEL you may hit problems if you are using commas to represent decimal points.* In the following example the `-1` switch in the call to `xlsread` tells MATLAB that we want to select data interactively, before pressing *OK* and returning to MATLAB.

```
>> data=xlsread('IO_example5.xls',-1)
```


Microsoft Excel - IO_example5 [Read-Only]
File Edit View Insert Format Tools Data Window Help
Arial 10
A2
Sheet1 Sheet2 Sheet3
Ready
Sum=-2.8317
NUM
Data Selection Dialogue
OK

	A
1	Header
2	-0.4326
3	-1.6656
4	0.1253
5	0.2877
6	-1.1465

Select data region in Excel worksheet.
Click OK to continue in MATLAB

Figure 5.6: *The xlsread function allows us to interactively select data in EXCEL and send it to MATLAB with a given variable name.*

MATLAB should read then output the information in the variable `data`:

```
data =
-0.4326
-1.6656
0.1253
0.2877
-1.1465
```

It is also possible to extract data automatically from an EXCEL sheet, this requires the desired cell position to be entered into the `xlsread` function. In this example we'll use the same EXCEL workbook, extracting data from cells A2 to B6 in the worksheet "Sheet2".

```
>> data=xlsread('IO_example5.xls','Sheet2','A2:B6 ')
data =
0.1139 0.2944
1.0668 -1.3362
0.0593 0.7143
-0.0956 1.6236
-0.8323 -0.6918
```

Again, this only demonstrates the most basic functionality of `xlsread`. When importing your own data read the documentation carefully.

5.1.5 NetCDF files

For those of you who work with large data sets, for example output from climate models or satellite observations, you will probably want to load data from NetCDF files into MATLAB. Importing from NetCDF into MATLAB is not built into MATLAB but a number of different functions have been written by the scientific community and are available for free download from various web sites. In my experience the best package is the *NetCDF Toolbox* which is available from **http://mexcdf.sourceforge.net/**. This package will provide you with routines for reading, modifying and creating NetCDF files.

5.2 Export

Most of the methods for exporting data from MATLAB parallel closely the import routines, so we won't dwell on them in too much detail. We'll look at three basic cases:

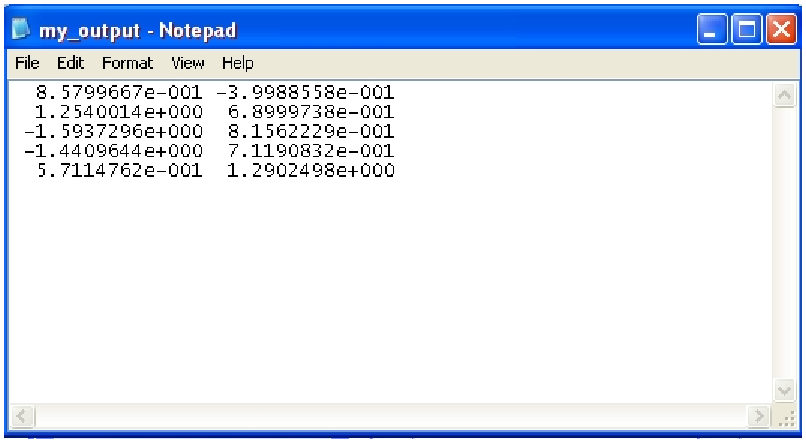
- Outputting a simple ASCII file.
- Exporting to an EXCEL workbook.
- Writing formatted text files.

5.2.1 Outputting a simple ASCII file

This can be done using the `save` function with an extra switch that tells MATLAB to save the data in ASCII rather than its own binary format. To use `save` we therefore need to define the output filename, the name of the variable to be saved and the output format.

```
>> y = randn(5,2) %generate some random numbers
y =
0.8580 -0.3999
1.2540 0.6900
-1.5937 0.8156
-1.4410 0.7119
0.5711 1.2902

>> save my_output.txt y -ASCII
```

The output file *my_output.txt* now contains the numbers which were stored in `y`.



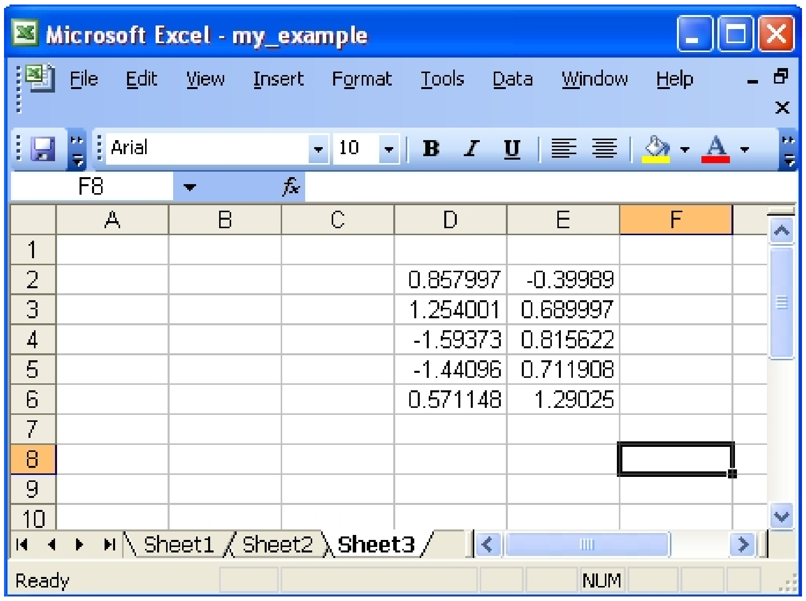
```
 8.5799667e-001 -3.9988558e-001
 1.2540014e+000  6.8999738e-001
-1.5937296e+000  8.1562229e-001
-1.4409644e+000  7.1190832e-001
 5.7114762e-001  1.2902498e+000
```

Figure 5.7: *Example output ASCII file generated by the save function.*

5.2.2 Exporting to an EXCEL workbook

The `xlswrite` function works in a very similar way to `xlsread`. We must define the workbook to which a given variable should be written (if it doesn't exist it will be created), the worksheet and range of cells for the output. For this example we'll make a new EXCEL workbook called **my_output.xls** and place the data from variable `y` (created above) in "Sheet3" over the cells spanning D2 to E6.

```
>> xlswrite('my_output.xls',y,'Sheet3','D2:E6 ')
```


Microsoft Excel - my_example
File Edit View Insert Format Tools Data Window Help
Arial 10
F8
Sheet1 Sheet2 Sheet3
Ready NUM

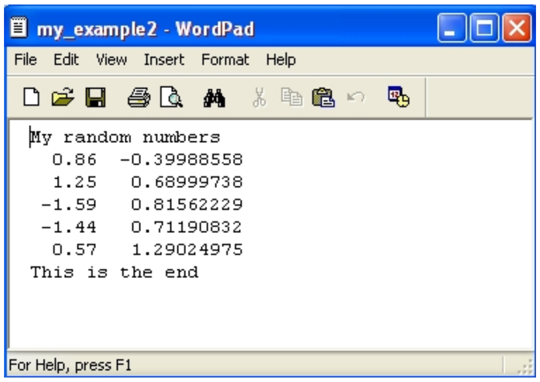
	A	B	C	D	E	F
1						
2				0.857997	-0.39989	
3				1.254001	0.689997	
4				-1.59373	0.815622	
5				-1.44096	0.711908	
6				0.571148	1.29025	
7						
8						
9						
10						

Figure 5.8: *Example output EXCEL file generated by the xlswrite function.*

5.2.3 Writing formatted text files

The inbuilt MATLAB I/O routines can be employed to generate output text files with well defined structures, including features such as header lines. This generally involves opening a file using the `fopen` function and writing lines of data or text one by one with specific formatting using the function `fprintf`. When using `fopen` we need to provide an additional argument `'w'`, which tells MATLAB to create a new file if one does not already exist with that name. In order to control the formatting we must tell MATLAB where to place carriage returns (using the `\n` symbol) and define the format of the numbers using field width and precision specifications. In the following example we'll output the first column using the format `%6.2f` (a maximum field width of 6 characters with 2 after the decimal place) and the second column with `%12.8f` (a maximum field width of 12 characters with 8 after the decimal place).

```
>> fname='my_example2.txt'
>> fid=fopen(fname,'w')
>> fprintf(fid,'My random numbers \n')
>> for i=1:5
   fprintf(fid, '%6.2f %12.8f \n', y(i,:));
end
>> fprintf(fid,'This is the end \n')
>> fclose(fid)
```



```
My random numbers
  0.86  -0.39988558
  1.25   0.68999738
 -1.59   0.81562229
 -1.44   0.71190832
  0.57   1.29024975
This is the end
```

Figure 5.9: *Example of formatted output using the fprintf function.*

5.3 Data Management

MATLAB is particular useful for managing and storing large data sets. Specific values from specific variables can be recalled quickly with a single statement rather than the tiring mouse work which is required with spreadsheets. In order to manage our data in a well organized manner we can use so-called *structural* arrays which allow data of varying types and sizes to be stored in *fields*. The fields form a hierarchy within the structure with each field representing a property or attributes that we want to associate data with. Using fields we can build up layers of data in a logical manner. To see how this would work we'll now look at an example, building up a structural array to contain the information for the sediment core MD962094. The information we want to store concerning the core is as follows.

1. Position (latitude, longitude, water depth)
2. The ship name and date the core was recovered
3. Available data (in this case depth and the color reflectance at 400 and 700 nm)

The first thing we need to decide on is a name for the structural array, it would seem sensible to use the name of the core: `MD962094`. The first information we'll input is the `waterdepth` of the core (giving both the value 2280.0 and the units "m"), which we will store in a higher level field called `position`. Notice how the path through the hierarchy of fields is controlled by the dot operator.

```
>> MD962094.position.waterdepth.value=2280.0
>> MD962094.position.waterdepth.units='m'
```

Note that when inputting the units a text string is used to store characters rather than numbers. We can now interrogate the variable at different levels again using the dot operator:

```
>> MD962094
MD962094 =
position:  [1x1 struct]

>> MD962094.position
```

```
ans =
waterdepth:  [1x1 struct]

>> MD962094.position.waterdepth
ans =
value:  2280
units:  'm'

>> MD962094.position.waterdepth.value
ans =
2280
```

Entering the remainder of the data follows the same pattern and as before the names that we use are arbitrary (they could be anything but is is always best to choose something meaningful). First we'll finish the position information by including the latitude and longitude:

```
>> MD962094.position.latitude.units='degrees'
>> MD962094.position.latitude.value=-20.0
>> MD962094.position.longitude.units='degrees'
>> MD962094.position.longitude.value=9.265
```

Next we'll enter the ship name and the date the core was recovered:

```
>> MD962094.ship = 'Marion Dufresne'
>> MD962094.date = 'Y1996 M10 D13'
```

Finally we can enter the colour reflectance data itself (we'll only enter the first few values to save space on the page) in each case defining both the values of the data and their units:

```
>> MD962094.data.reflectance.depth.units='m'
>> MD962094.data.reflectance.depth.value=[0.05;0.1;0.15]
>> MD962094.data.reflectance.nm400.units='%'
>> MD962094.data.reflectance.nm400.value=[19.65;21.44;18.40]
>> MD962094.data.reflectance.nm700.units='%'
>> MD962094.data.reflectance.nm700.value=[35.26;36.35;34.08]
```

If you are working with date sets from a number of a different cores then you can store their data in a series of structural variables with a consistent structure, then it is simple to call precisely the data you want for any given core. Lets imagine that we want to find the maximum value of the 400 nm reflectance data:

```
>> max(MD962094.data.reflectance.nm400.value)

ans =

36.3500
```

5.4 Exercise: Entering data into a structural array

In this exercise I would like you construct a structural array which contains the details and values of a real data set. You can choose a data set from your own research or alternatively look on the PANGAEA database to find a simple data set that you can work with. You can use the example from the previous section to give you an idea of how to build up the array and use the I/O methods we discussed above to import the data into MATLAB.

Chapter 6

Plotting in two-dimensions

6.1 Plotting in two-dimensions

MATLAB can be used to produce a wide variety of plots thanks to its comprehensive graphics libraries. The appearance of plots can be controlled from the command line, which means that you can include automatic plotting in your script files and M-functions. Many more specialized types of graphics which are not included in the distributed version of MATLAB are available on the File-Exchange.

6.1.1 Figure windows

To create a figure window into which a plot can be placed simply give the command:

```
>> figure
```

This will produce a blank window. At this point it is important to realize that the figure windows has certain properties, called *Figure Object Properties*, which we may want to change in order to modify the overall appearance of our final graphic. We can look at these properties using the `get` function with the `gcf` specifier which corresponds to the *handle* of the current figure.

```
>> get(gcf)
```

The text listed on the screen gives the figure's properties on the left and their current state on the right. For example, lets look at the color of the figure, we can either look through the returned list or request only the color property using the `get` function:

```
>> get(gcf,'color') %request only the color property

ans =
0.8000 0.8000 0.8000
```

The returned values correspond to the RGB value of the current figure color (values can range between 0 and 1, where [0 0 0] would be black and [1 1 1] would be white). If we want to change the color of the figure, for example to red we use the `set` function and provide the appropriate color vector (in this case [1 0 0]):

```
>> set(gca,'color',[1 0 0]) %reset the color property
```

Now when you look at the figure it should be red (a set of axes may also have appeared). MATLAB figures and plots are all controlled by adjusting the properties of the handles, just as in the example we performed.

6.1.2 Using Handles

At first the use of handles can seem like a very complicated way of controlling the appearance of graphics compared to software such as EXCEL where you just need to double-click to change items. The advantage of using handles is that we can include adjustments to our figures and plots in script files, so with a bit of programming the appearance of a graphic can be optimized automatically. Understanding and using handles takes some time, especially remembering the specific property names which you may want to adjust. Over time you will remember the important handle properties and for the more obscure ones the MATLAB help system can tell you what they do and how they can be adjusted. Throughout the following examples we will be using handles to modify graphics, so it is important to be familiar with the following simple specifiers.

- `gcf` specifies the current figure handle
- `gca` specifies the current axes handle
- `gco` specifies the current graphic object handle

6.2 Simple data plotting

Now we'll generate some simple *x*,*y* data and then plot it using the `plot` function:

```
>> t = linspace(0,3*pi,30); %30 point equally spaced vector between 0 and 3pi
>> x = sqrt(t).*cos(t); %calculate the x values
>> y = sqrt(t).*sin(t); %calculate the y values
>> figure %create a new figure window
>> plot(x,y) %plot the x,y data
```

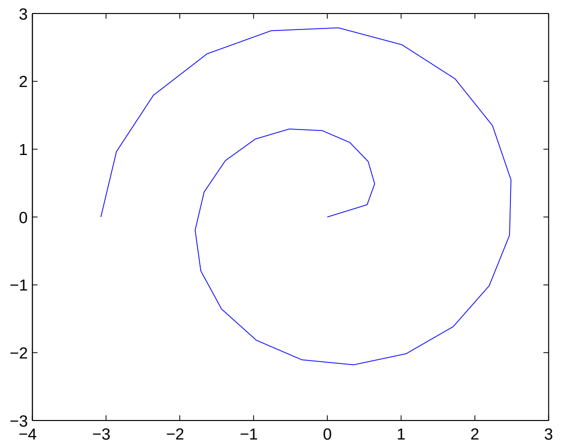


Figure 6.1: *A simple two-dimensional plot of x,y data.*

The `plot` function has a number of additional arguments that allow us to control the style of the plotted line. For example if we wanted to plot the line in red we would use the additional option:

```
>> plot(x,y,'r') %plot the x,y data in red
```

You should now see that the plot uses a red line. We can also specify plot symbols rather than a line, for example open circles in green:

```
>> plot(x,y,'og') %plot the x,y data as green open circles
```

or small closed symbols in black:

```
>> plot(x,y,'.k') %plot the x,y data as black points
```

or diamond symbols (specified using `'d'`) in magenta with a connecting line:

```
>> plot(x,y,'dm-') %plot the x,y data as magenta diamonds with a line
```

or the same as above with a dashed line:

```
>> plot(x,y,'dm--') %plot the x,y data as magenta diamonds with a dashed line
```

As mentioned above, you will memorize these various specifiers with time, but they are all listed in the MATLAB help so it is very easy to find them. If we want to store the handle for the plotted line we need to provide an output variable for the plot function, we can then view the handle using `get` and modify its properties with `set`.

```
>> h=plot(x,y,'dm--') %output the handle as h
>> get(h)
```

Listed on the screen you'll see the properties of the plotted data, for example you can see the `LineStyle` is `'--'` just as it was set in the plot command and the `Marker` is `'diamond'`. As an example of modifying the handle we'll set the color to red, the symbols to squares, the size of the symbols to 12 points, the lines to be dot-dashed and to have a thickness of 2 points.

```
>> set(h,'color',[1 0 0]) %set the color to red
>> set(h,'marker','square') %set the symbols as squares
>> set(h,'markersize',12) %set the symbol size to 12 pts
>> set(h,'linestyle','-.') %set the line to dot-dashed
>> set(h,'linewidth',2) %set the line thickness to 2 pts
```

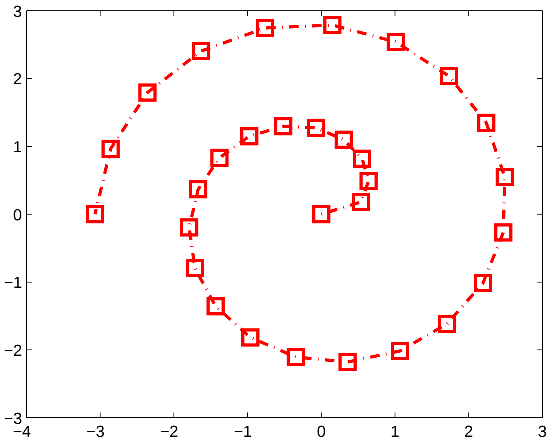


Figure 6.2: *A two-dimensional plot of x,y data, the appearance of which has been changed using its handle.*

At anytime we can plot additional data onto the graphic. Now we'll generate a second data set and add it to the plot.

```
>> x1=-y; %create a new x array
>> y1=-x; %create a new y array
>> h1=plot(x1,y1,'g') %plot the data as a green line with the handle h1
```

When we look at the figure we can see that there is a problem, the old line has been removed from the plot and replaced with the new line. If we want to show both lines on the same plot we need to use the `hold on` command, which tells MATLAB to add data to the graphic rather than replacing it.

```
>> h0=plot(x,y,'k') %plot the data as a black line with the handle h0
>> hold on %tells MATLAB we want to add to the plot
>> h1=plot(x1,y1,'g') %plot the data as a green line with the handle h1
```

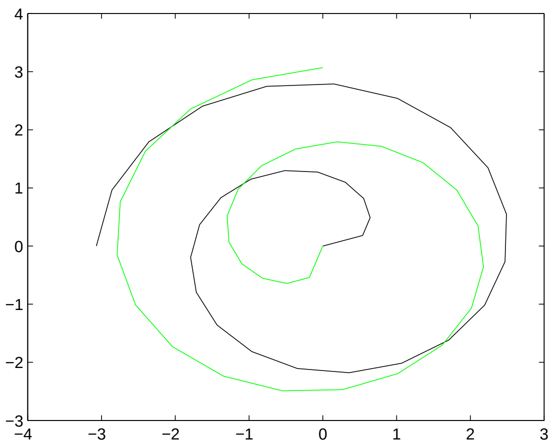


Figure 6.3: *The two sets of x,y data plotted together using the hold on command.*

We also have handles for each plotted data set, `h0` and `h1`, so we could modify either of the lines using the `set` command.

6.2.1 Modifying the axes

So far we have used the handles of the plotted lines to change their appearance, but we can also take a similar approach with the appearance of the axes. There are some simple commands which allow us to add labels and a title, which must be defined as character strings.

```
>> xlabel('X data') %add a text string to the x-axis
>> ylabel('Y data') %add a text string to the x-axis
>> title('Two spirals demonstrating the plot command') %add a title
```

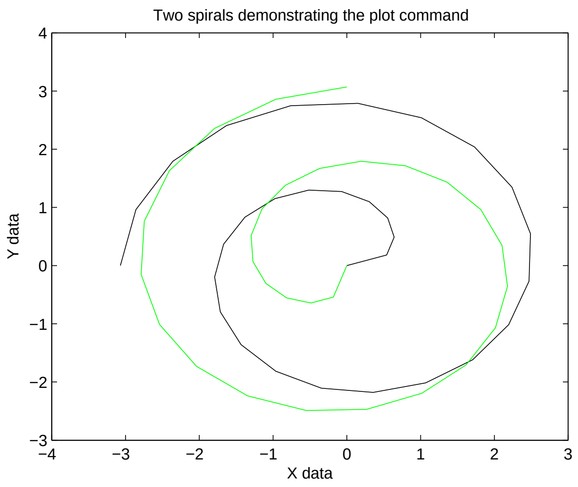


Figure 6.4: *The two sets of x,y data plotted with axis labeling and a title.*

We can access the handle for the axes using the `gca` identifier.

```
>> get(gca)
```

As you can see there are a large number of properties and we can change any of them using the `set` command. As with the other handles it is important to check the MATLAB documentation to find what the allowable states for any given property are. The following sets of commands show some of the properties that can be changed, enter them one by one each time checking the figure to see how things have changed.

```
>> set(gca,'tickdir','out') %tick marks point outwards rather than inwards
>> set(gca,'yminortick','on') %plot intermediate ticks on the y-axis
>> set(gca,'fontsize',16) %set the text size to 16 points
>> set(gca,'xdir','reverse') %reverse the direction of the x-axis
>> set(gca,'ylim',[-2 2]) % set the y-axis range as -2 to 2
>> set(gca,'yaxislocation','right') %place the y-axis on the right hand side
>> set(gca,'xgrid','on') %plot a grid over the x-axis
>> set(gca,'fontangle','italic') %use italic fonts
>> set(gca,'xtick',[-4 -2 0 1 2.5 3.7]) %plot xticks at specific values
```

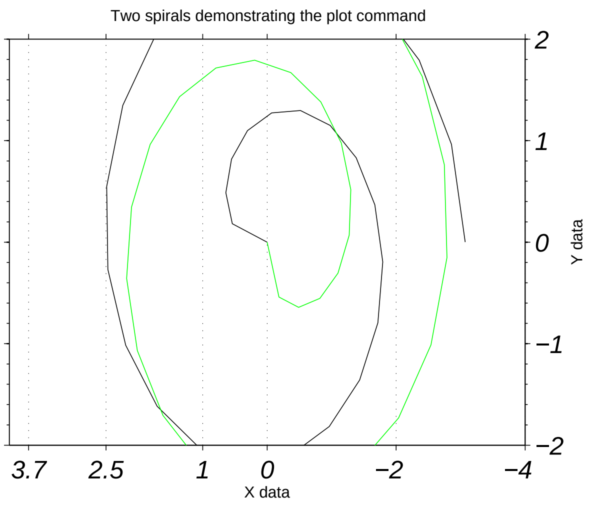


Figure 6.5: *The two sets of x,y data plotted with a collection of modifications to the axes.*

As you can see you can change every aspect of a graphic using the handles, all you need to know is the name of the property you want to change.

6.2.2 Multiple sets of axes in a single figure

Often you will want to plot and compare a number data sets within a single figure window, this can be done using the `axes` command or its extension `subplot`. When we create a new figure window we can place axes in a specific position with a specific size. The `axes` command has 4 numerical inputs entered as a position vector `axes('position', RECT)`, where `RECT` = [left, bottom, width, height] specifies the location and size of the side of the axis box, relative to the lower-left corner of the Figure window, in normalized units where (0,0) is the lower-left corner and (1.0,1.0) is the upper-right. In this first example we'll make a figure with two sets of axes and plot our spiral data separately.

```
>> close all %close the existing figure windows
>> figure %create a new figure window
```

Now we'll form a set of axes which have their bottom left corner located at (0.1,0.2) a width of 0.3 and a height of 0.7. We'll then plot the `x`,`y` data inside the axes.

```
>> axes('position',[0.1 0.2 0.3 0.7]) %create specific axes
>> plot(x,y,'b') %plot x and y in blue
```

Lets add a second set of axes which have there bottom left corner located at (0.5,0.4) a width of 0.4 and a height of 0.2. Then we'll plot the data set `x1`,`y1` inside the axes.

```
>> axes('position',[0.5 0.4 0.4 0.2]) %create specific axes
>> plot(x1,y1,'r') %plot x1 and y1 in red
```

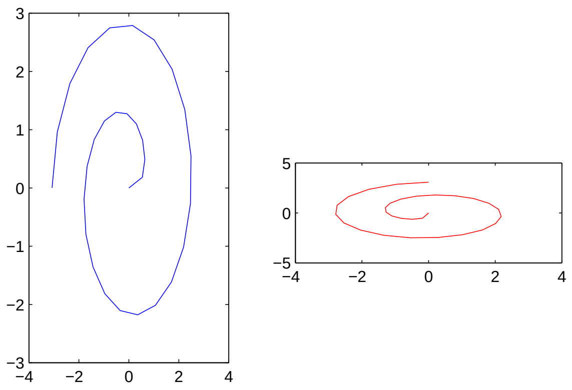


Figure 6.6: *User specified axes within a single figure window.*

If you would like to create a set of regularly sized and spaced axes then you can use the `subplot` command. `subplot` requires 3 inputs, the number of rows of axes, the number of columns of axes and which set of axes it should make active. This may sound confusing but hopefully the example will make things clear, we'll construct 6 sets of axes in a grid with 2 rows and 3 columns. When we specify the active axes (the third input of `subplot`) they are numbered in a sequence which runs from left to right along each row from top to bottom. To demonstrate this concept we'll use the axes out of their natural sequence. Remember to keep looking at the figure to see how it evolves with each command.

```
>> close all %close the existing figure windows
>> figure %create a new figure window
>> subplot(2,3,1) %2 rows & 3 columns of axes, use position 1
>> plot(rand(20,1),rand(20,1),'o') %plot some random data
>> title('2 x 3 form, position 1')

>> subplot(2,3,5) %2 rows & 3 columns of axes, use position 5
>> plot(rand(20,1),rand(20,1),'o') %plot some random data
>> title('2 x 3 form, position 5')
```

As with our previous examples, we can output the handle for a specific set of axes created by `subplot` and use it to modify the axes properties.

```
>> h3=subplot(2,3,3) %2 rows & 3 columns of axes, use position 3, return the handle

>> plot(rand(20,1),rand(20,1),'o') %plot some random data
>> title('2 x 3 form, position 3')
>> set(h3,'color',[1 0 0]) %set these axes to be red
```

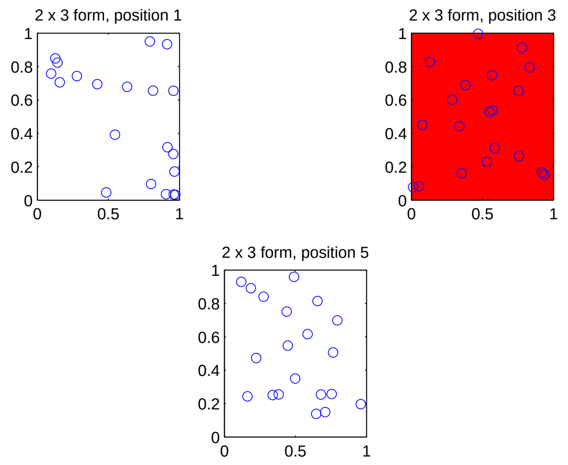


Figure 6.7: *A regular sequence of axes formed using the subplot command.*

6.2.3 Different types of plots

So far we have only used the `plot` command, but MATLAB can make many different types of 2D plots. All these plotting functions work in a similar way (use `help` or the online documentation to understand how each is used), form a basic plot of the data and then customize the graphic using the handles to modify specific properties. Shown below is a graphic displaying the different types of plots, there are many more on the File-Exchange and of course you can even write your own plot functions.

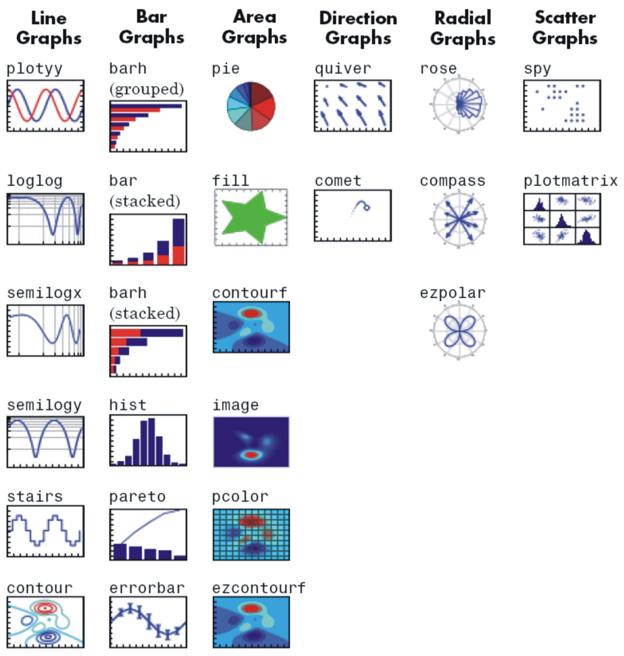


Figure 6.8: *The various 2D plot types available in MATLAB*

6.2.4 Contour plots

One specific type of plot that as geoscientists we must pay attention to is the contour plot. MATLAB has inbuilt functions to plot contour maps, but at a first glance the form of the input data can appear to be a bit confusing. To contour spatial data we must of course have two location coordinates, for example latitude and longitude, and some measured parameter such as sea-surface temperate. Most often we collect these kind of data as a series of *[x,y,z]* triplets, forming three columns of data. Unfortunately MATLAB does not follow this system (not surprisingly we need to use a matrix form for data input), so we have to perform some preprocessing before we can contour data. As an example of the form in which MATLAB can plot x,y,z data we'll use `peaks` which is a function of two variables, obtained by translating and scaling Gaussian distributions. We'll generate and plot the peaks function for 20 x 20 data grids.

```
>> clear all, close all
>> n=20; %number of grid points
>> [x,y,z]=peaks(n); %evaluate peaks on a 20 x 20 grid
>> size(x) %we can see the size of x

ans =

20 20

>> contour(x,y,z,10) %plot data with 10 contours
>> colorbar %add a colorbar to show the z values
```

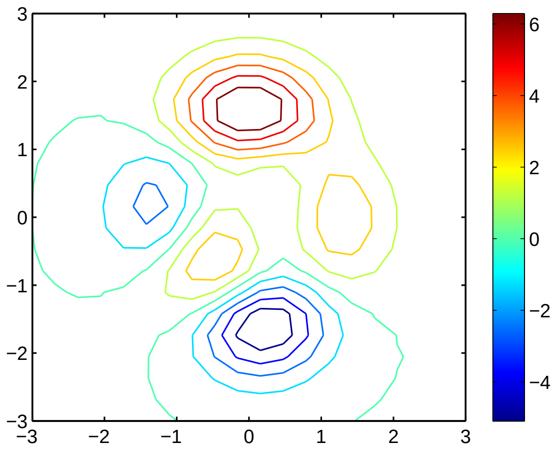


Figure 6.9: *Contour plot of the peaks function evaluated on a 20 x 20 grid.*

To understand the form of the matrices MATLAB uses as inputs for its contouring functions we'll evaluate the `peaks` function on a smaller 4 x 4 grid.

```
>> clear all, close all
>> n=4; %number of grid points
>> [x,y,z]=peaks(n); %evaluate peaks on a 4 x 4 grid

>> x
x =

-3 -1 1 3
-3 -1 1 3
-3 -1 1 3
-3 -1 1 3

>> y
y =

-3 -3 -3 -3
-1 -1 -1 -1
1 1 1 1
3 3 3 3
```

This shows us how the inputs of the spatial coordinates are organized. The horizontal coordinate, `x` in this case, has columns of constant value and rows with varying values, while the vertical coordinate, `y`, shows the opposite pattern (basically `x` and `y` are defining a grid of locations). Not only do the inputs for the contouring routines need to be in matrix form but they must be equally spaced in both directions, you can see in the example above that the spacing is equal to 2 units. So before we can contour our data we must interpolate our collection of *[x,y,z]* triplets on to an equally spaced grid of x and y values (this is the same as the *gridding* step in SURFER). As an example of how to grid data and plot it we will work with a simple topographic data set which is stored in the file dmap.mat with the variables `x`, `y` and `z`.

```
>> clear all, close all
>> load dmap %load the data file
>> plot(x,y,'or') %plot the positions of the measured data points
>> xlabel('x coordinate') %label the x axis
>> ylabel('y coordinate') %label the y axis
```

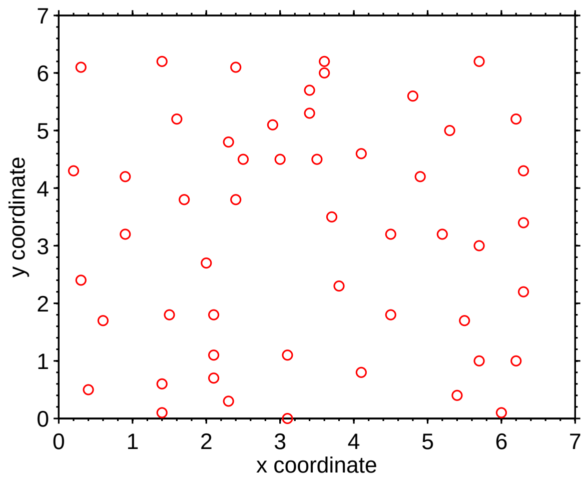


Figure 6.10: *Positions of the data points in the dmap data set*

The first thing to note is that our data is in columns not matrices and that the points were not measured on a regular grid. Now we must create some regular grids onto which we can interpolate the data. To do this we'll use the `meshgrid` function which takes equally spaced vectors and forms them into grids.

```
>> xi=[0:0.25:6.5]; %equally spaced row vector of x positions
>> yi=[0:0.25:6.5]; %equally spaced column vector of y positions
>> [xi_grid,yi_grid]=meshgrid(xi,yi);
>> plot(xi_grid,yi_grid,'.k') %plot the grid positions
```

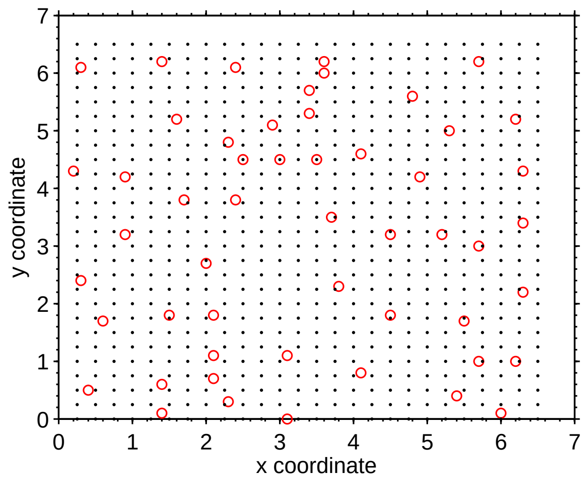


Figure 6.11: *Positions of the data points in the dmap data set (red circles) with the new positions forming the regular grids (black dots)*

We can now see the positions of the regular spaced grid points and our final task is to interpolate our topographic data onto these new grids (we'll discussed interpolation in detail later in the course). To perform the interpolation we use the function `griddata`, giving our measured data `x`, `y` and `z` and the uniform grids `x_grid` and `y_grid` onto which the data should be interpolated.

```
>> zi_grid=griddata(x,y,z,xi_grid,yi_grid); %interpolate the z data
>> hold on
>> contour(xi_grid,yi_grid,zi_grid,10) %plot 10 contour lines
```

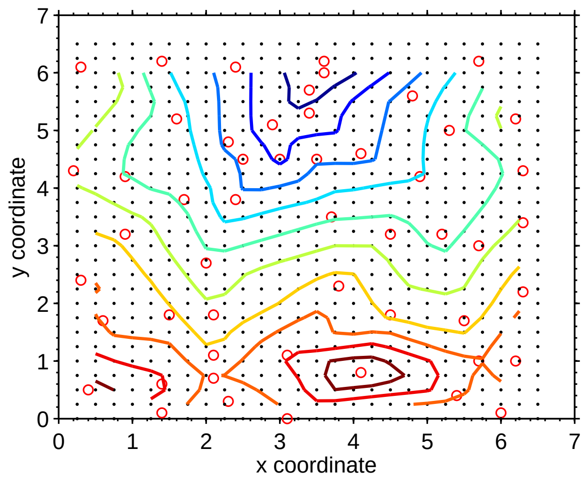


Figure 6.12: *Contours determined from the data points in the dmap data set (red circles) after interpolation onto the new positions in the regular grids (black dots)*

There are a variety of different contour-type maps available, for example `contourf` will produce a filled contour map and `pcolor` plots a colored grid. Try the following examples (note that MATLAB does not extrapolate the data so some regions of the maps will be empty, again we'll talk about this later).

```
>> figure
>> contourf(xi_grid,yi_grid,zi_grid,10) %plot 10 filled contour intervals
>> figure
>> pcolor(xi_grid,yi_grid,zi_grid) %pseudocolor plot
```

6.2.5 Saving and exporting figures

MATLAB figures can be saved from the drop-down menu in the specific figure window. This means you can reload figures and continue working on them at a later time. You can also export the figures from the drop-down menu into various graphic formats. I've always found it useful to export figures as encapsulated postscript (**.eps*), which can then be imported into software such as CorelDraw and modified (encapsulated postscript produces vector graphics).

6.3 Exercise: Life's ups and downs in the Logistic Map

In this exercise we'll combine a number of techniques that we have studied in the course so far to generate a data set and produce a high quality plot of the final output. If you image a population within an enclosed environment, for example goldfish in a garden pond, there are a number of factors which will control how the population grows or shrinks from one generation to the next. For example:

- Food supply
- Sexual reproduction
- Disease
- Predation
- Competition with other species
- ... and many more

A number of simple theories have been formulated to consider how the population will change from one generation to the next, but they rely on unrealistic assumptions such as unlimited food supply and free reproduction. When ecologists studied real animal populations they found a number of different patterns as a function of time. Using a value of 1 to represent the maximum possible population and 0 to represent extinction, three basic patterns of "stable", "periodic" and "random" behavior were defined.

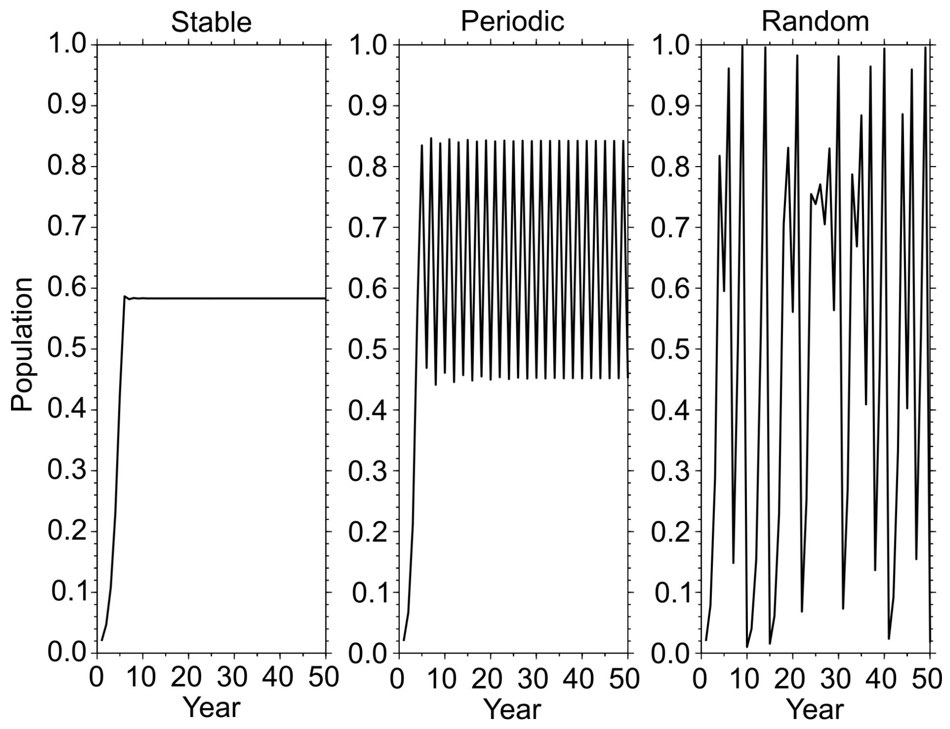


Figure 6.13: *A "stable" population moves towards a constant equilibrium value. Alternatively, "periodic" populations oscillate between two or more values. Finally "random" populations reveal no coherent pattern.*

Ecologists believed that two possible mechanisms could have been responsible for the observed patterns. If the population is in equilibrium with the environment then "stable" and "periodic" patterns could be expected. However if the population is heavily influenced by unpredictable environmental factors the behavior of "random" populations could be explained. These two systems are however very different, so which is correct?

In the 1950's ecologists developed the so-called "logistic equation", which was designed to model population behavior, it contains a term to express population growth from one generation to the next and a second term which restricts the population when it becomes too large.

$$x_{next} = r * x_{initial} * (1 - x_{initial}) \tag{6.1}$$

where $x_{initial}$ is this year's population, x_{next} is next year's population and r is the "biotic potential". The biotic potential influences how much the population changes from one generation to the next, it will be controlled by factors such as food supply, competition, etc. The term $(1 - x_{initial})$ keeps the growth within the defined limits because as x increases, $(1 - x)$ decreases. Some key features of the logistic equation are:

- When x is low the population grows rapidly.
- At intermediate levels of x the population growth is almost zero.
- At high levels of x the population crashes.

Using the logistic equation is is possible to produce "stable", "periodic" and "random" behavior simply by changing the constant, r, in the equation, therefore we have one mechanism to describe all three types of behavior. With increasing computing power, it became possible to study the dynamics of the Logistic equation. The investigation is very simple.

1. At random choose a starting population value $0 < x < 1$
2. At random choose a value of r in the range $0 < r < 4$
3. Calculate the final population after a large number of years (say 2000)
4. Plot a point on a chart at position (r, x_{final})

When this procedure is repeated a large number of times to map the behavior of the logistic equation we produce the so-called logistic map.

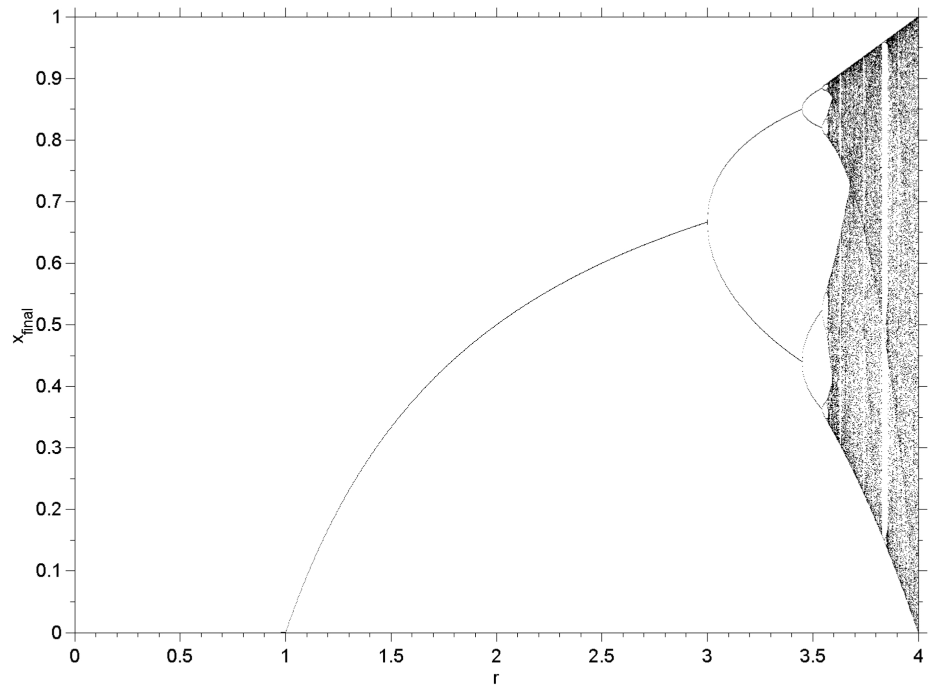


Figure 6.14: *The Logistic Map.*

Different regions of the map tell us different things about the behavior of a population which obeys the logistic equation. With r in the range $0 < r <\sim 1$ the population will always become extinct, no matter what the starting size of the population is.

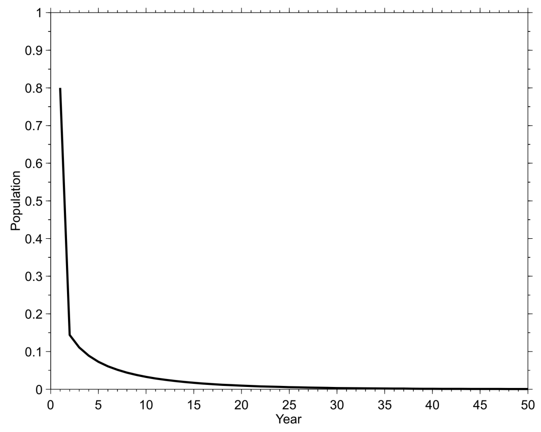


Figure 6.15: *The evolution of a population with* $x_{initial}$ *= 0.8 and* r *= 0.9*

With r in the range $\sim 1 < r <\sim 3$ the population reaches a steady single value no matter what the starting value of $x_{initial}$ is.

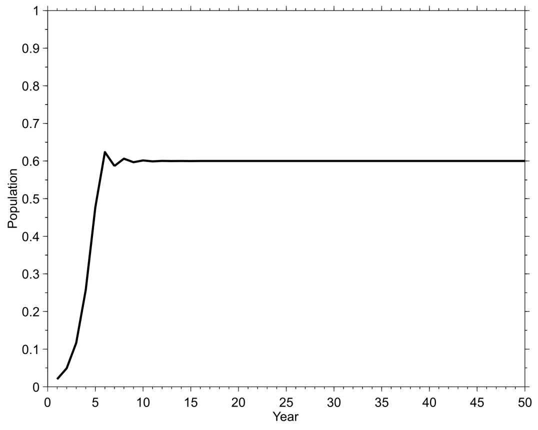


Figure 6.16: *The evolution of a population with $x_{initial}$ = 0.02 and r = 2.5*

With r in the range $\sim 3 < r <\sim 3.42$ the population shifts between two values. This is called "period two" behavior.

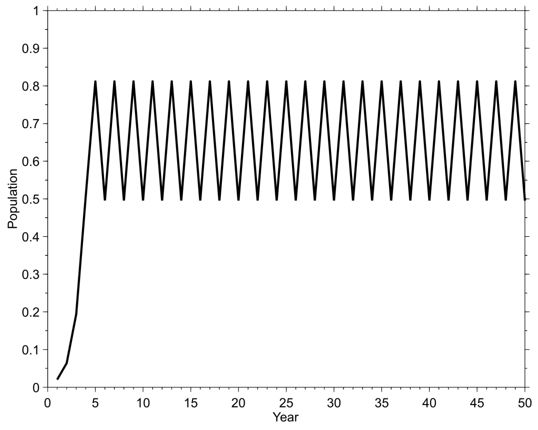


Figure 6.17: *The evolution of a population with $x_{initial}$ = 0.02 and r = 3.25*

With r in the range $\sim 3.42 < r <\sim 3.57$ the population shifts between four values. This is called "period four" behavior.

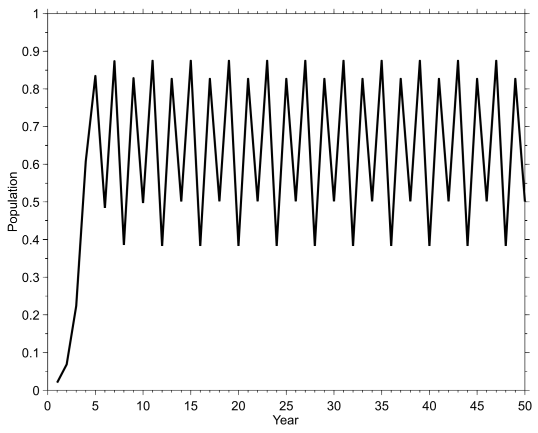


Figure 6.18: *The evolution of a population with* $x_{initial}$ *= 0.02 and* r *= 3.5*

With r in the range $\sim 3.57 < r <\sim 4$ the population shifts between many values. This is called "chaotic" behavior.

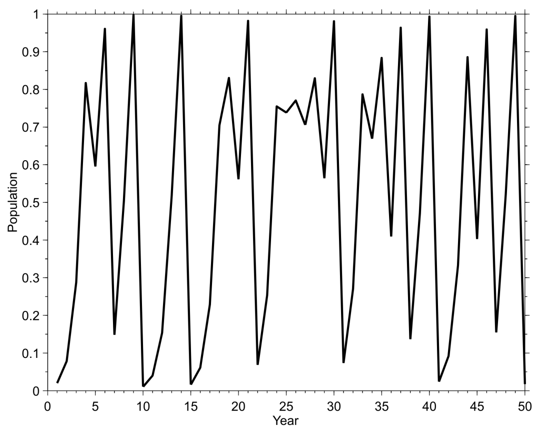


Figure 6.19: *The evolution of a population with* $x_{initial}$ *= 0.02 and* r *= 3.99*

Your task is to generate your own version of the logistic map.

Steps to completing the exercise

This exercise will require a number of different steps.

- Generate code to iterate the logistic equation from a given starting point for a given number of time steps.
- Simulate a large number of populations with different values of r and $x_{initial}$.
- Extract data from your model runs to form a plot of the logistic map.
- Use your plotting skills to make a nice version of the logistic map.

The simplest way to approach this problem is to write a script which will allow you to generate one time series and then extend the code to run from multiple starting points. To run for a single value of r and $x_{initial}$:

1. Generate a random number between 0 and 1 to provide $x_{initial}$.
2. Generate a random number between 0 and 4 to provide r.
3. Use a `for` loop to iterate the logistic equation 2000 times, updating the population after each iteration.
4. Find the final value of the population after 2000 iterations, this will give you one point (r,x_{final}) for the logistic map.

Once you have a method for finding a single point you can then repeat it thousands of times to find a collection of x_{final} values for different values of r and $x_{initial}$.

Functions you may find useful

- `rand` - Uniformly distributed pseudo-random numbers.

Chapter 7

Plotting in three-dimensions

7.1 Plotting in three-dimensions

Three-dimensional graphics are a natural extension of the two-dimensional graphics we studied earlier. We can plot data points and lines in three-dimensional space using the function `plot3`. It is also possible to plot surfaces and meshes, where the data input must be on regular grids, which means interpolation of your data may be necessary. As with the two-dimensional graphics the appearance of the three-dimensional plots is controlled via the graphic handles.

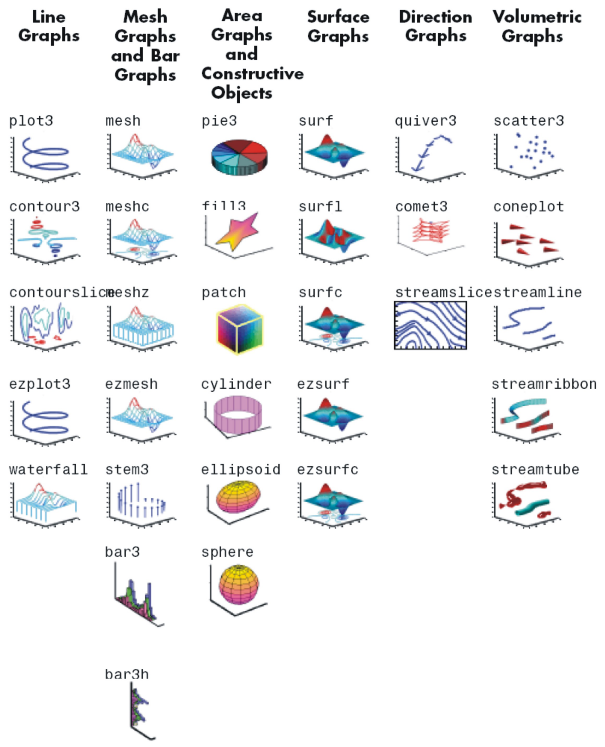


Figure 7.1: *The various 3D plot types available in MATLAB*

7.1.1 Plotting points and lines

The function `plot3` works in exactly the same way as the two-dimensional `plot` function we used earlier. Of course `plot3` requires one more input which corresponds to the value of the third coordinate (defined as z in MATLAB). In the following example we will load data from a Lorenz system, which consists of time (variable t) and the trajectories x, y, z. First we will load the data and plot the separate components in two-dimensions.

```
>> clear all, close all
>> load lorenz

>> subplot(3,1,1)
>> plot(t,x,'r') %plot x as a red line
>> ylabel('x')

>> subplot(3,1,2)
>> plot(t,y,'g') %plot y as a green line
>> ylabel('y')

>> subplot(3,1,3)
>> plot(t,z,'b') %plot z as a blue line
>> ylabel('z')
>> xlabel('t')
```

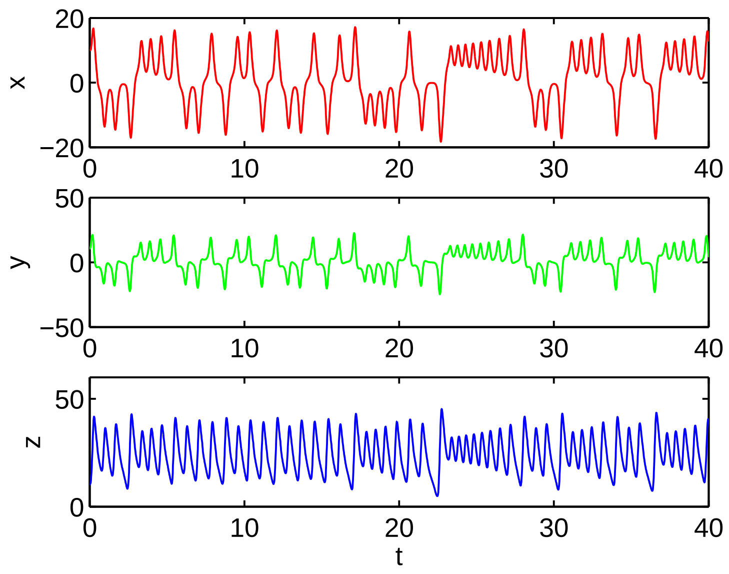


Figure 7.2: *Time series for the Lorenz trajectories in x, y and z.*

Now We'll create a new figure and and then plot the trajectories together in three dimensions.

```
>> figure
>> plot3(x,y,z) %plot trajectories together in 3D
>> xlabel('x trajectory') %label the x-axis
>> ylabel('y trajectory') %label the y-axis
>> zlabel('z trajectory') %label the z-axis
>> title('Trajectories for the Lorenz system')
```

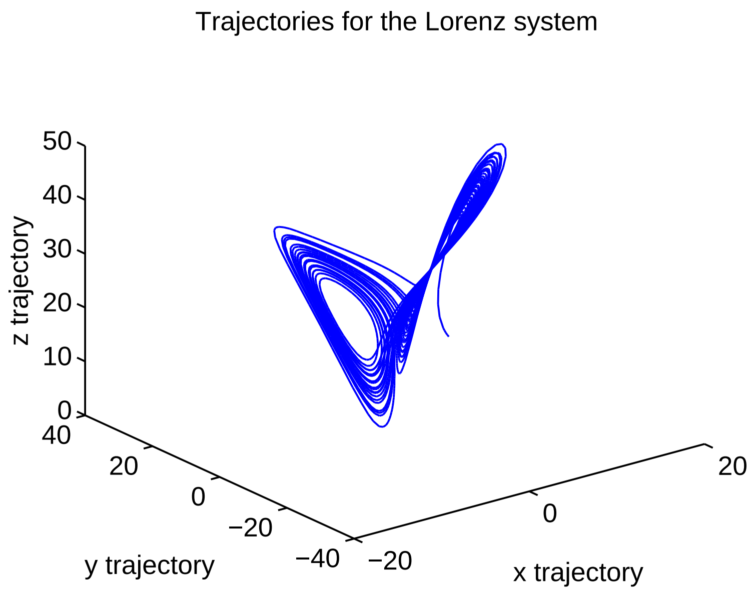


Figure 7.3: *Lorenz trajectories in x, y and z.*

We've now done the basics but we can use the graphics handles to improve the appearance of the plot. In particular the default orientation of the plot (termed the camera position) does not provide the best view of the data structure (you can rotate the data manually using the *Rotate3D* button). First we'll add grids to plot box and then modify the property CameraPosition using the axes handle to provide a better view of the data.

```
>> set(gca,'xgrid','on','ygrid','on','zgrid','on') %plot grids
>> set(gca,'cameraposition',[200 -200 200]); %change the orientation
```

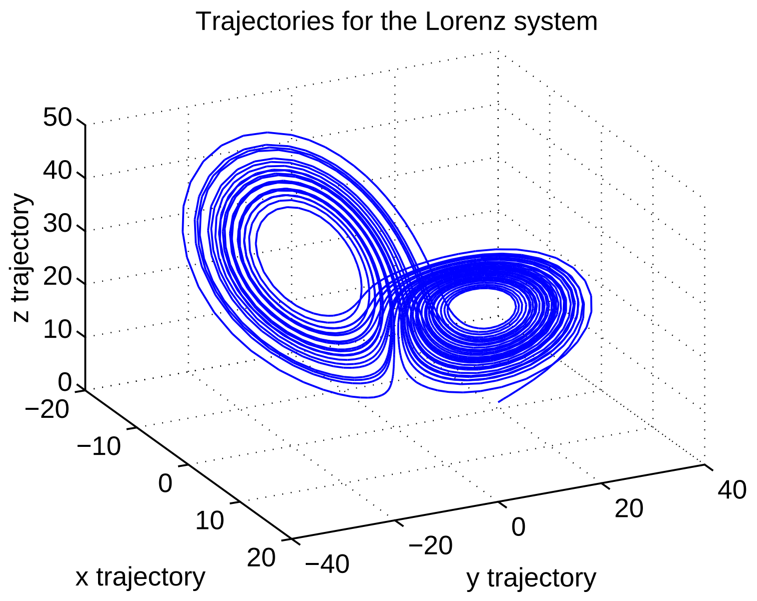


Figure 7.4: *Lorenz trajectories in x, y and z, plotted with grids and rotated to provide a clear view of the data. Note the values for the CameraPosition were obtained by gradual modification until a satisfactory position was found.*

7.1.2 Surface plots

Earlier we used two-dimensional contour maps to represent topographic data, but MATLAB also has the capacity to represent such data in a three-dimensional manner using surfaces and meshes. As with our contour data the input for such plots must be composed of regular grids, so we need to preprocess our data using `meshgrid` and `griddata`.

```
>> clear all, close all
>> load dmap %load the data file
>> xi=[0:0.25:6.5]; %equally spaced row vector of x positions
>> yi=[0:0.25:6.5]; %equally spaced column vector of y positions
>> [xi_grid,yi_grid]=meshgrid(xi,yi);
>> zi_grid=griddata(x,y,z,xi_grid,yi_grid); %interpolate the z data
```

Now we'll create a surface plot of the data, add the original data on top, rotate the surface and make it slightly transparent. Feel free to experiment with the handle of the surface (which will be called `hs`) to see what effects you can create.

```
>> h=surf(xi_grid,yi_grid,zi_grid) %create a surface plot
>> hold on %add to the plot
>> hs = plot3(x,y,z,'ok','markerfacecolor','k') %original data points
>> set(h,'facealpha',0.5) %surface is semitransparent
>> xlabel('x') %label the x-axis
>> ylabel('y') %label the y-axis
>> zlabel('Height') %label the z-axis
>> set(gca,'cameraposition',[-21.5 44.2 2691]) %rotate the plot
>> colorbar %add a colorbar
```

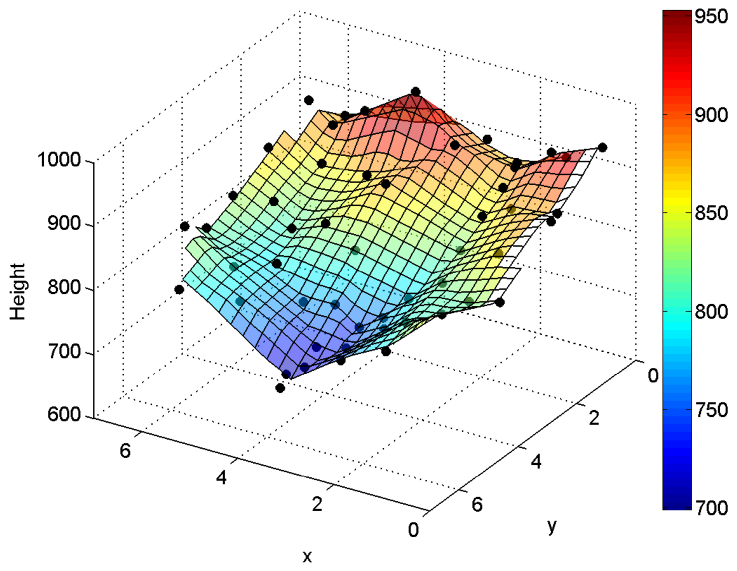


Figure 7.5: *Topographic data set plotted as a surface.*

By default MATLAB uses a sequences of colors, termed a *colormap*, which runs from blue through to red. However there are others to choose from and you can even create your own, for example there is an inbuilt colormap called `gray` which uses a grayscale, this can be implemented in the current figure using the `colormap` command.

```
>> colormap gray
```

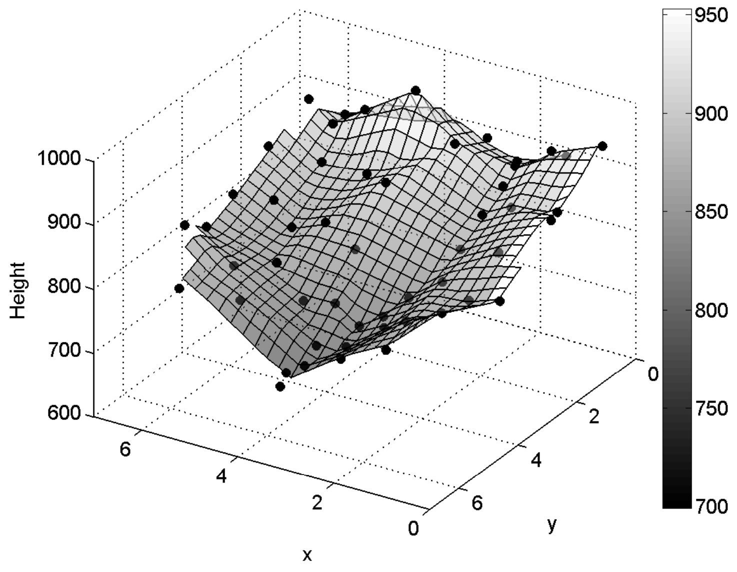


Figure 7.6: *Topographic data set plotted as a surface with the gray colormap.*

There are a variety of different ways in which we could have represented the topographic data in the example, see figure 7.1

7.1.3 Overlaying images

To demonstrate the flexibility of the graphics routines in MATLAB we'll now look at a more complicated example, that involves plotting a satellite image of the earth on top of a sphere. We will read the satellite image (in the form of a jpg) directly from a internet location and store it as a matrix.

```
>> clear all, close all
>> filename = 'http://veimages.gsfc.nasa.gov/2430/land_ocean_ice_2048.jpg';
>> cdata = imread(filename); %read the image from the given location
>> size(cdata) %size of the image data

ans =

1024 2048 3
```

Note that the size of the image data is 1024 rows (latitude), 2048 columns (longitude) and 3 layers. Each layer provides information on a given color channel, red, green and blue. Now we will produce a collection of data points which lie on a sphere, which will form the basis of our Earth:

```
>> [x, y, z] = sphere(72); %generate data points on a sphere
>> figure %open a new figure window
>> h = surf(x,y,z) %plot a surface from the sphere data
>> axis vis3d %set the axis perspective suitable for 3D
```

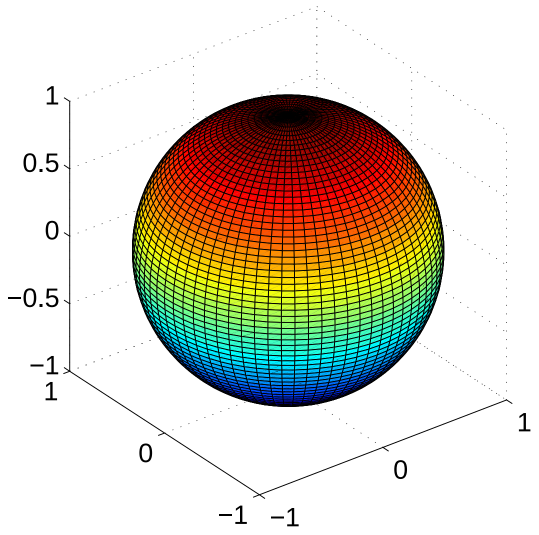


Figure 7.7: *Surface plot of the data points produced by the sphere function. We will replace the colors of the panels with those from the downloaded satellite image.*

As a final step we use the handle of the plotted surface (our sphere) to replace the existing colors of the panels with the colors obtained from the downloaded jpg (which are now stored in the variable `cdata`):

```
>> set(h, 'facecolor', 'texturemap') %prepare the panels to be colored
>> set(h,'cdata', cdata); %color the panels with the jpg data
>> set(h,'edgecolor', 'none'); %remove lines between panels
>> set(gca,'visible','off'); %hide the axes box
>> set(gcf,'color',[0 0 0]); %set the background color to black
```

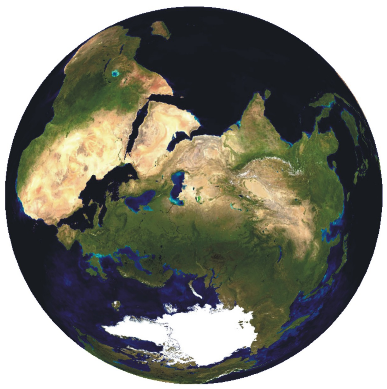


Figure 7.8: *Surface plot of the data points produced by the sphere function with the colors of the panels based on the downloaded satellite image.*

You can now use the *Rotate3D* and *Zoom* buttons to navigate around your Earth.

7.2 Exercise: Plotting elevation data

In the file *Hawaii.mat* you'll find bathymetric and topographic data for region around the main Hawaiian Islands. The data represent a scatter of latitude and longitude points with their respective height data. The aim of this exercise is to produce regular latitude and longitude grids onto which the height data can be interpolated and then to plot surfaces which represent the topography and bathymetry.

1. Grid the data at a resolution of 0.1^o for both the longitude and the latitude.
2. Plot the gridded data using a suitable 3D plot, add appropriate labels etc.
3. Plot a version of the data that includes the points above sea-level and has a height of zero everywhere else.
4. Plot a version of the data that includes the points below sea-level and has a height of zero everywhere else.

Steps to completing the exercise

This exercise requires a number of steps which follow the previous examples in this chapter, creating surfaces which represent the bathymetry and topography can be achieved using relational operators. The basic steps of the exercise are as follows:

- Load the data file *Hawaii.mat* which contains the variables `latitude`, `longitude` and `elevation`.
- Create a regular array of latitudes between the minimum and maximum values in the data set with a spacing of 0.1^o.
- Create a regular array of longitudes between the minimum and maximum values in the data set with a spacing of 0.1^o.
- Produce regular grids of latitude and longitude from the prepared arrays.
- Interpolate the height data onto the regular position grids.
- Create a copy of the interpolated data with all the values less than zero replaced with zeros. This data can be plotted as a surface representing the topography.
- Create a copy of the interpolated data with all the values greater than zero replaced with zeros. This data can be plotted as a surface representing the bathymetry.

Functions you may find useful

- `min` - Smallest elements in array.
- `max` - Largest elements in array.
- `meshgrid` - Generate X and Y arrays for 3-D plots.
- `griddata` - Data gridding.
- `find` - Find indices of elements which meet relational criteria.
- `surf` - 3-D shaded surface plot.
- `xlabel` - Label x-axis.
- `ylabel` - Label y-axis.
- `zlabel` - Label z-axis.

Chapter 8

Interpolation

Interpolation is a method by which we can use data points to estimate the unknown value of a function at a given location. This is a common problem in the geosciences, for example, if we have a collection of calibrated radiocarbon ages through a sediment core, how can we estimate the age of the sediment at any given depth? MATLAB has inbuilt functions to perform interpolation in any given number of dimensions, they are easy to use but it is important to appreciate that the various options you can select from can strongly influence your results.

8.1 Interpolation in one dimension

This is the simplest form of interpolation, estimating a value y' for a given value of x' based on a collection of known x, y data points. As an example of the different methods available in MATLAB we will load an x, y data series and perform different interpolation tasks.

```
>> clear all %clear MATLAB's memory
>> close all %close all figures

>> load bedf %load the x,y data
>> plot(x,y,'.r') %plot the data as red points
>> xlabel('x'), ylabel('y') %label the axes
```

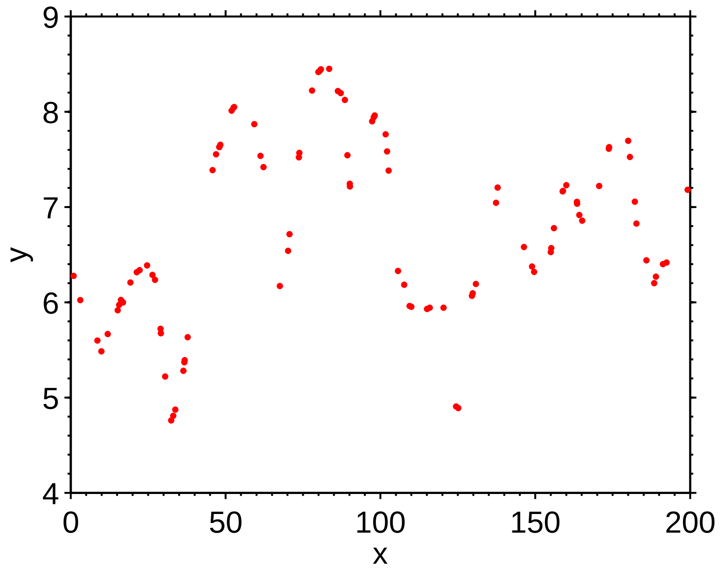


Figure 8.1: *plot of the bedf data set.*

Given the measured data we can start to ask questions, such as, what value of y could we expect at the location $x' = 41$. One simple way to answer this question is using "linear" interpolation, where we determine the equation for a straight line between the two data points which span $x' = 41$. Once we know the equation we can estimate the value of y'.

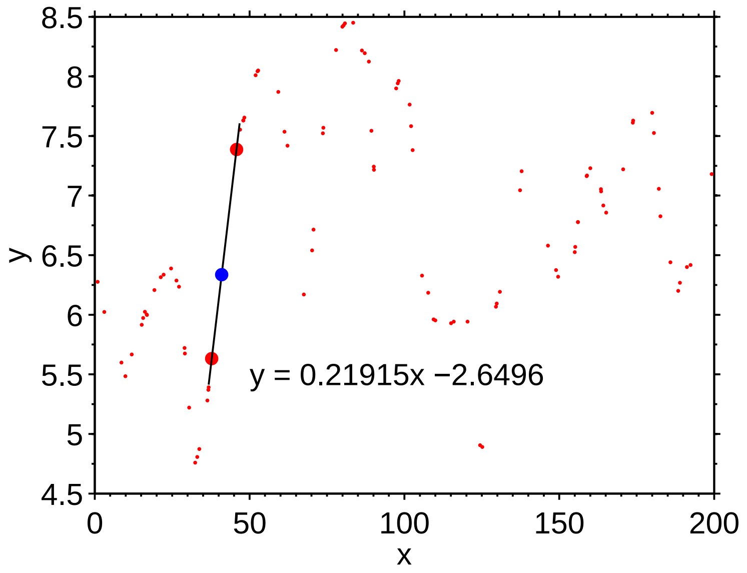


Figure 8.2: *If we are interested in finding the value of y' at x = 41, we simply determine the straight line between the two data points which span the neighborhood of interest (large red symbols). Once we have the equation;* $y = 0.21915x - 0.26496$ *we can estimate that when* $x' = 41$, $y' = 6.34$ *(blue circle).*

8.1.1 Piecewise Linear Interpolation

The full name of the approach we have just discussed is "piecewise linear interpolation", because the data is split into a series of consecutive pieces and each one is fit with a linear function. The task can be performed simply in MATLAB using the function `interp1`, the information we need to supply is the measured x data (first argument), the measured y data (second argument), the x' value for the interpolation (third argument) and the interpolation method to use, which in this case is `'linear'`.

```
>> y0=interp1(x,y,41,'linear') %using the (x,y) data make a linear estimate of
y at x=41

y0 =
6.3357
```

It is important to note that when performing interpolation the x-data must be distinct, in other words you can't have any repeat points on the x-axis. MATLAB will return an error if you supply repeat x-values, so in some cases it is necessary to prepare your data before processing. We'll test this using indexing in order to enter the first values in x and y into the `interp1` function twice.

```
>> y0=interp1(x([1 1:end]),y([1 1:end]),41,'linear')
 ???  Error using ==> interp1 at 261
The values of X should be distinct.
```

When performing interpolation with the `interp1` function we don't need to limit ourselves to determining single values. If we supply the function with a vector of x' values it will perform the interpolation for each one. This is very useful for a lot of time series analysis techniques which assume that you have equally spaced data. If your measured data isn't equally spaced then you can simply interpolate it to produce a suitable new data series. Now we'll work with the same x, y data as above, but interpolate it onto equally spaced x'-locations in the range $0 \leq x' \leq 200$ and with a interval in x' of 1 unit.

```
>> figure %produce a new figure
>> plot(x,y,'or','markerfacecolor','r') %plot the original data
>> hold on
>> x0=[0:1:200]'; %generate the equally spaced x array
>> y0=interp1(x,y,x0,'linear'); %interpolate y onto x0
>> plot(x0,y0,'.b-')%plot the linear interpolation
>> xlabel('x'), ylabel('y') %label the axes
```

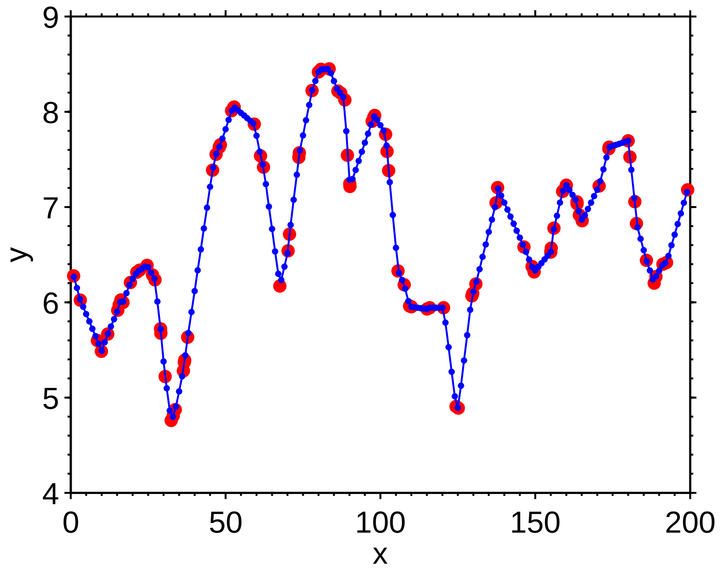


Figure 8.3: *Linear interpolation of unequally spaced x,y data onto a new equally spaced x vector.*

When we examine the interpolated values we discover that for locations at the start and end of the interpolated data MATLAB has returned `NaN` (Not a Number), rather than a real number.

```
>> [x0(1:3),y0(1:3)] %list the first 3 values of the interpolated data

ans =

0 NaN
1.0000 6.2677
2.0000 6.1500

>> x(1:3) %the first value of the x input data

ans =

0.9268
3.0807
8.6048
```

We can now see that the interpolation at $x' = 0$ is not possible because it is outside the range of our input data. We can attempt to overcome this problem using *extrapolation*, which makes the assumption that the properties at the edges of the x,y data continue unmodified outside the measured range. Applying extrapolation to `interp1` is simple, but as we will see later the results have to be treated with extreme caution.

```
>> y0=interp1(x,y,x0,'linear','extrap'); %add a 5th option to perform extrapolation

>> [x0(1:3),y0(1:3)] %we now have predicted y-values outside of the measured x-range
```

```
ans =

0 6.3854
1.0000 6.2677
2.0000 6.1500
```

So far we have only looked at *linear* interpolation, which assumes that the gaps between data points can be connected by straight lines. There are however a number of different approaches built into `interp1` that we can use for interpolation and they all give different results! It is important to realize at this stage there is no single correct method with which to approach interpolation, each method does different things depending on how it is designed. All the methods are "wrong" in so far as (unless we are very lucky) they will not predict y' correctly, but they are wrong in different ways depending on their methodology. The best we can do is try to choose the method of interpolation which is most suitable for the type of data set we are working with.

8.1.2 Nearest Neighbor Interpolation

The *nearest neighbor* method is probably the simplest approach to interpolation. If we are interested in the value of y' at a location x' we simply search for the measured data point which has the x-value closest to our point of interest and assign y' to the same value as this data point. We will work again with the `bedf` data from the previous example.

```
>> y0=interp1(x,y,x0,'nearest'); %interpolate y onto x0 using the nn approach
>> figure %produce a new figure
>> plot(x,y,'or','markerfacecolor','r') %plot the original data
>> xlabel('x'), ylabel('y') %label the axes
>> hold on
>> plot(x0,y0,'.g-') %plot the nearest-neighbor interpolation
>> title('Nearest neighbor interpolation')
```

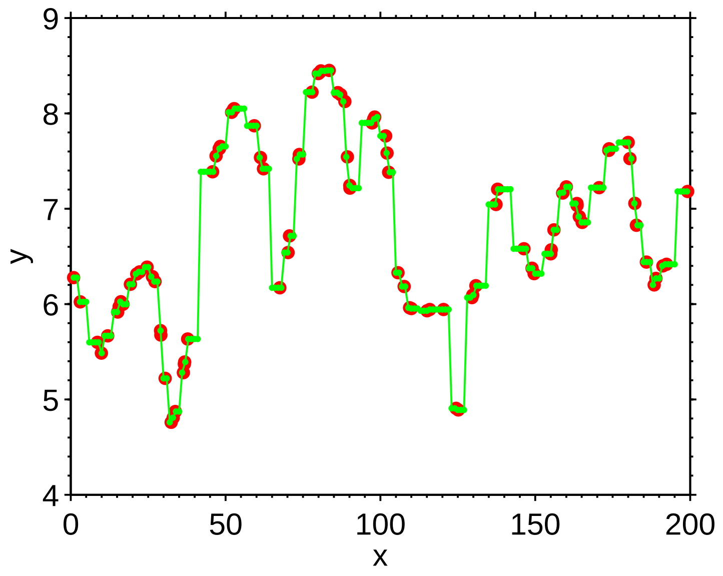


Figure 8.4: *Interpolation of unequally spaced x,y data (red symbols) onto a new equally spaced x' vector using the nearest neighbor approach (shown in green).*

8.1.3 Cubic spline Interpolation

The *cubic spline* method is an attractive approach to interpolation because it allows the lines connecting the measured data points to be curved. It also has advantages concerning its derivatives, which we will look at quickly.
Linear interpolation preserves continuity of the fitted function $f(x)$, i.e. there are no breaks in the curve, each data point is connected by two straight lines to its neighbors. For a data point at a given location in x, the straight lines which connect the point to it's neighbors have the same value of y when evaluated at x. The same cannot be said for the derivatives of the two straight lines which may have different values at x and therefore are not continuous. This leads to sudden jumps in the curve and the interpolation is not smooth. In contrast a cubic spline fits a series of cubic functions between neighboring data points which are constrained to have continuous first and second derivatives. Again, for a given data point with location x this means that the cubic functions which link it to its left and right neighbors must have identical values of their first and second derivatives at x. This constraint makes the fitted functions very smooth. The mathematics of how to fit a cubic spline is very elegant and amazingly efficient, if you are interested in a more detailed explanation of cubic splines look at chapter 3 of Moler (Numerical Computing with MATLAB).

```
>> y0=interp1(x,y,x0,'spline'); %interpolate y onto x0 using a cubic spline
>> figure %produce a new figure
>> plot(x,y,'or','markerfacecolor','r') %plot the original data
>> xlabel('x'), ylabel('y') %label the axes
>> hold on
>> plot(x0,y0,'.k-') %plot the cubic spline interpolation
>> title('cubic spline interpolation')
```

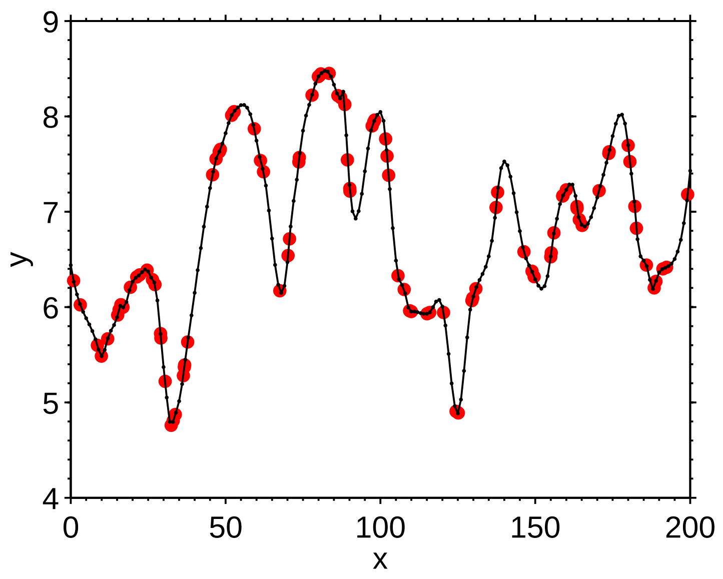


Figure 8.5: *Interpolation of unequally spaced x,y data (red symbols) onto a new equally spaced x vector using a cubic spline (shown in black).*

As we can see in Figure 8.5 the cubic spline can produce local extremes in the interpolation which cannot always be justified by the measured data, for example look at the peak at $x = 180$. Cubic splines are said to "overshoot" data when they form a spurious peak and "undershoot" when they form a spurious trough.

8.1.4 PCHIP Interpolation

The Piecewise cubic Hermite method (pchip) provides a balance between the linear approach (which produces no curvature) and the cubic spline (which can produce artifical extremes). The pchip approach is a so-called "shape-preserving" spline method which exams the derivatives of neighboring cubic functions to determine if together they will form a peak or a trough (this will be the case if their derivatives have different signs). Where peaks or troughs will appear special precautions are taken to make sure that an overshoot or undershoot does not occur. Again, you can look at chapter 3 of Moler (Numerical Computing with MATLAB) for more details concerning the pchip algorithm.

```
>> y0=interp1(x,y,x0,'pchip'); %interpolate y onto x0 using pchip
>> figure %produce a new figure
>> plot(x,y,'or','markerfacecolor','r') %plot the original data
>> xlabel('x'), ylabel('y') %label the axes
>> hold on
>> plot(x0,y0,'.b-') %plot the pchip interpolation
>> title('Piecewise cubic Hermite interpolation')
```

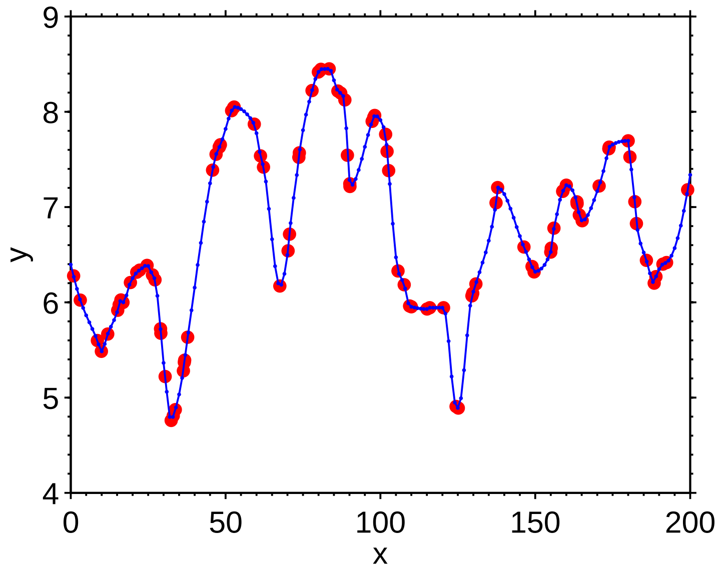


Figure 8.6: *Interpolation of unequally spaced x,y data (red symbols) onto a new equally spaced x vector using the pchip method (shown in blue).*

As you can see each interpolation method produces a different result. As discussed above it is important not to think of right and wrong in this situation. Each method is right in so far as it produces a function according to its defined constraints, but they are also all wrong because they will never form perfect predictions of y' except under very special circumstances (i.e. when the function the used in the interpolation matches exactly to the form of the function that the data follows).

8.2 The dangers of extrapolation

We now going to generate a simple time series which follows a sine function. We will then continue the time series using extrapolation based on the different techniques we discussed above. This will hopefully convince you that extrapolation should be performed with extreme caution.

```
>> x=[0:1:96] % selection of x data sites
>> y=sin(2.*pi.*x./25) %sine function with a period of 25 units
>> plot(x,y,'.r-') %plot the data
>> set(gca,'xlim',[0 106]) %extend the graphic window
>> x0=[96:1:110]'; %x locations for extrapolation
>> y0=interp1(x,y,x0,'linear','extrap'); %linear extrapolation
>> hold on
>> plot(x0,y0,'b') %add linear extrapolation to the figure
>> y0=interp1(x,y,x0,'nn','extrap'); %nn extrapolation
>> plot(x0,y0,'g') %add nn extrapolation to the figure
>> y0=interp1(x,y,x0,'spline','extrap'); %cubic spline extrapolation
>> plot(x0,y0,'k') %add spline extrapolation to the figure
>> y0=interp1(x,y,x0,'pchip','extrap'); %pchip extrapolation
>> plot(x0,y0,'m') %add pchip extrapolation to the figure
>> legend('data','linear','nn','spline','pchip',0) %add a legend
```

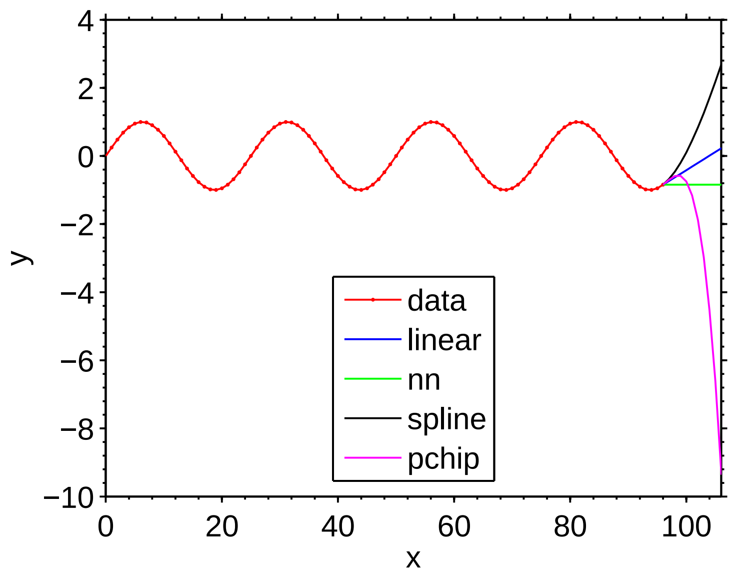


Figure 8.7: *Extrapolation of the sine function data (red symbols) with the different methods available in interp1. Note that the various methods give dramatically different results and none of them are close to approximating the continuing sine function.*

The dangers of extrapolation we put into a geoscience context 125 years ago by Mark Twain in his memoir *Life on the Mississippi* (1884):

"In the space of one hundred and seventy six years the Lower Mississippi has shortened itself two hundred and forty-two miles. That is an average of a trifle over a mile and a third per year. Therefore, any calm person, who is not blind or idiotic, can see that in the Old Olitic Silurian Period, just a million years ago next November, the Lower Mississippi was upwards of one million three hundred thousand miles long, and stuck out over the Gulf of Mexico like a fishing-pole. And by the same token any person can see that seven hundred and forty-two years from now the Lower Mississippi will be only a mile and three-quarters long, and Cairo [Illinois] and New Orleans will have joined their streets together and be plodding comfortably along under a single mayor and a mutual board of aldermen. There is something fascinating about science. One gets such wholesale returns of conjecture out of such a trifling investment of fact."

8.3 Interpolation in two dimensions

The ideas of interpolation in one-dimension can be extended into higher dimensions. MATLAB has a number of inbuilt functions for interpolation in two or more dimensions, but the one most suitable for most purposes is `griddata` which fits a grided surface to a collection of data points (basically the same task as performed in the software package GRAPHER). As before there are a number of different methods to perform the interpolations, simple ones such as "linear" and "nearest neighbor" and more complex ones, such as "cubic" and "v4" which allow the surface to be curved between the data points. To demonstrate the use of `griddata` we are going to load some topographic measurements, plot them as points in three-dimensions and fit different surfaces to them. Our data has to be interpolated on to regular grids, this can be done using the `meshgrid` function.

```
>> load topo %data containing the variables x, y, z
>> figure %create a new figure window
>> plot3(x,y,z,'ok','markerfacecolor','k') %plot the data as black points
>> xlabel('x'), ylabel('y'), zlabel('z') %label the axes
```

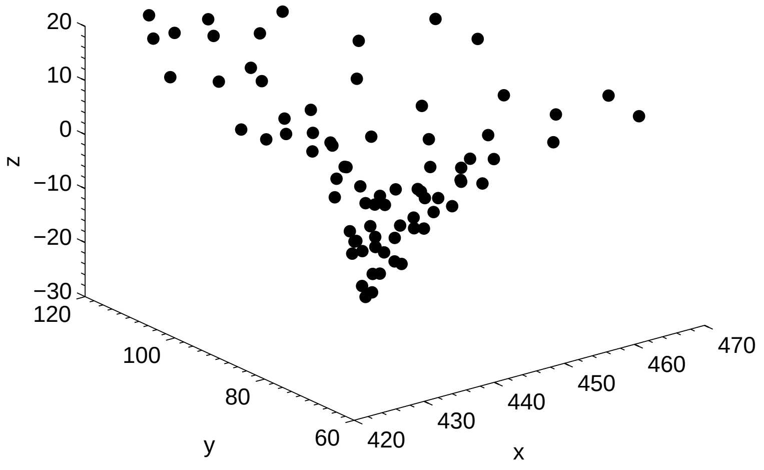


Figure 8.8: *Topographic raw data plotted as a collection of points. If you would like to explore the structure of the data you can use the "Rotate3D" button in the figure window.*

Now we must create the regular grids onto which the data will be interpolated. This is done by defining vectors in the x and y directions and bringing them together using the function *meshgrid*.

```
>> x0=[420:0.25:470]; %define x-locations for grid (201 points)
>> y0=[70:0.25:120]; %define y-locations for grid (201 points)
>> [XI,YI]=meshgrid(x0,y0); %generate x,y grids (201 x 201 points)
```

Once the grids are prepared the interpolation can be performed and the results plotted. The first three inputs for *griddata* are the `x`, `y`, `z` coordinates of the raw data, the next to inputs are the `XI` and `YI` grids onto which the data will be interpolated, and the final input is a string defining the interpolation method. First we'll use linear interpolation:

```
>> ZI=griddata(x,y,z,XI,YI,'linear'); %perform the linear interpolation
>> hold on
>> surf(XI,YI,ZI) %plot the interpolated surface
>> shading interp, light %smooth the surface appearance and illuminate
```

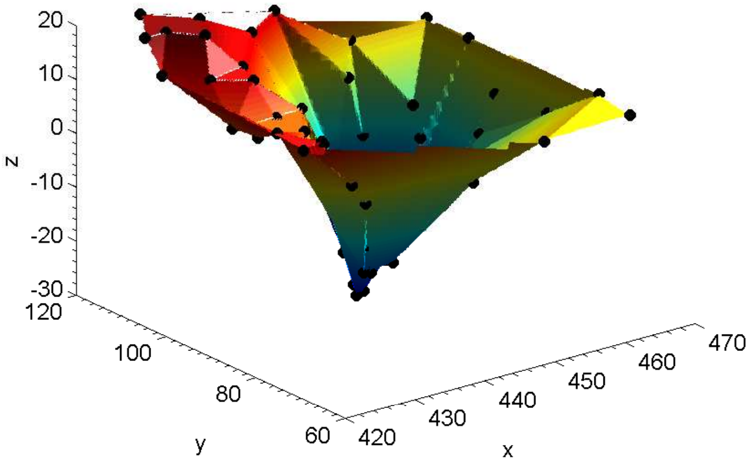


Figure 8.9: *Topographic raw data plotted as a collection of points with a colored surface fitted by linear interpolation. Note that all the faces joining the points are flat because the linear interpolation does not allow curvature.*

Now we'll replot the data and perform the interpolation using the "v4" method.

```
>> figure %create a new figure window
>> plot3(x,y,z,'ok','markerfacecolor','k') %plot the data as black points
>> xlabel('x'), ylabel('y'), zlabel('z') %label the axes
>> ZI=griddata(x,y,z,XI,YI,'v4'); %perform the v4 interpolation
>> hold on
>> surf(XI,YI,ZI) %plot the interpolated surface
>> shading interp, light %smooth the surface appearance and illuminate
```

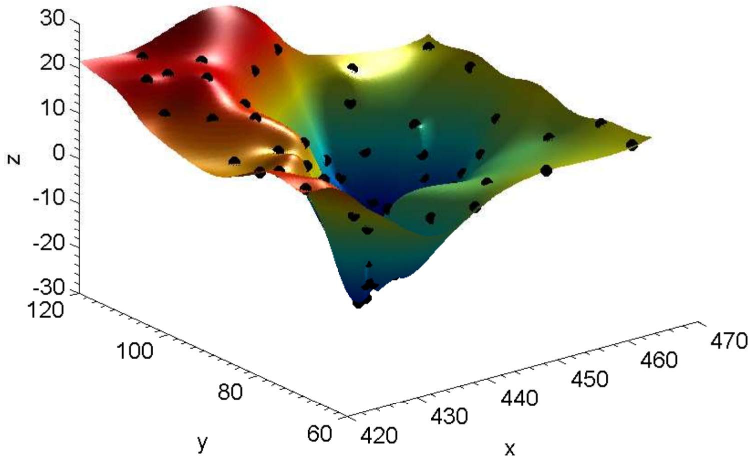


Figure 8.10: *Topographic raw data plotted as a collection of points with a colored surface fitted by "v4" interpolation.*

The "v4" method allows curvature and is smoother than the linear approach. As we discussed with interpolation in one-dimension, both techniques are wrong in that they cannot predict the values of z exactly, so we always have to think carefully about the interpolation method and what it actually represents.

8.4 Interpolation in higher dimensions

MATLAB can perform interpolation in higher dimensions (three upwards) using the function `ndgrid` to generate uniformly spaced data grids onto which `interpn` can interpolate data.

Chapter 9

Optimization

9.1 Minimization in one dimension

We are interested in finding the minimum value of the function:

$$f(x) = \frac{-1}{(x-0.3)^2+0.01} - \frac{1}{(x-0.9)^2+0.04} - 6 \tag{9.1}$$

If possible the best first step is to look how the function behaves over a range of different x values. To do this we must first write a MATLAB function to perform the calculation. This is simply:

```
function y=myfunction(x)
%Calculation of equation 9.1

y=-1./((x-0.3).^2+0.01)-1./((x-0.9).^2+0.04)-6;
```

Now we can calculate the value of equ. 9.1 for a series of x values.

```
>> x = linspace(-1,2,100); %100 equally spaced values between -1 and 2
>> y = myfunction(x); %call the function using the values in x
>> plot(x,y) %plot the result
>> xlabel('x') %add the x-axis label
>> ylabel('y') %add the y-axis label
```

Visual inspection shows us the the minimum is around $x \sim 0.3$. But we can use MATLAB to make a better evaluation, using the technique of **minimization**. The function `fminsearch` will numerically find the minimum value of a provided function from a predefined starting point (in other words our first guess for the value of x). The call to `fminsearch` is simple, you provide the name of the function you wish to minimize, the variable to use, and your starting point. In

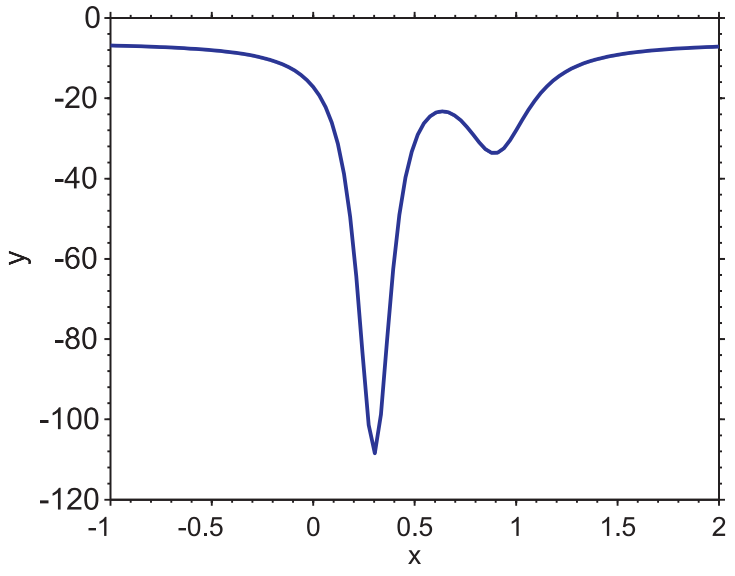


Figure 9.1: *Evaluation of equ. 9.1 in the range [-1,2].*

this case we will minimize `myfunction` with a starting value of $x = 0.5$

```
>> x0 = fminsearch(@(x) myfunction(x), 0.5)

x0 =

0.3004
```

The returned value tell us that within a predefined tolerance equ. 9.1 reaches its minimum value at $x = 0.3004$. We can add this information to the plot:

```
>> hold on
>> plot(0.5,myfunction(0.5),'or') % add a red maker at the starting point
>> plot(x0,myfunction(x0),'ob') % add a blue maker at the minimum
```

In order to minimize equ. 9.1 we wrote a separate function, however, if your expression is relatively simple you can also use an inline function:

```
>> f = inline('-1./((x-0.3).^2+0.01)-1./((x-0.9).^2+0.04)-6')
>> x0 = fminsearch(@(x) f(x), 0.5)

x0 =

0.3004
```

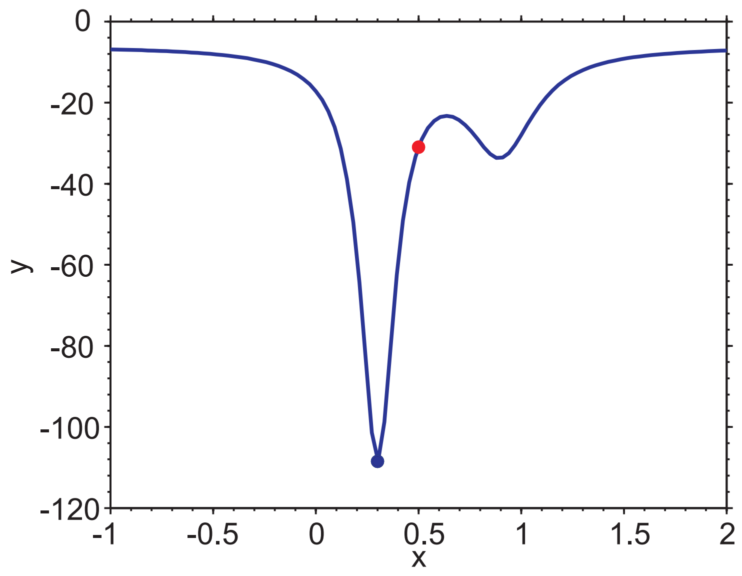


Figure 9.2: *Minimization of equ. 9.1, from the starting point $x = 0.5$. The starting point is shown as a red circle, the finishing point as a blue circle.*

9.1.1 How does it work?

There are a wide variety of different methods with which to perform minimization, each of which is suitable for different kinds or problems. One of the most commonly employed (and the simplest to understand) techniques is the so-called "gradient descent" method. To understand the ideas behind gradient descent we will use a zoom-up of our `myfunction` example. We can think of the shape of our function defined in equ. 9.1 as a hilly topography, if we placed a ball at position, $x = 0.5$, the ball would roll down the hill until is stops in the deepest part of the valley, which of course represents the minimum we are looking for.

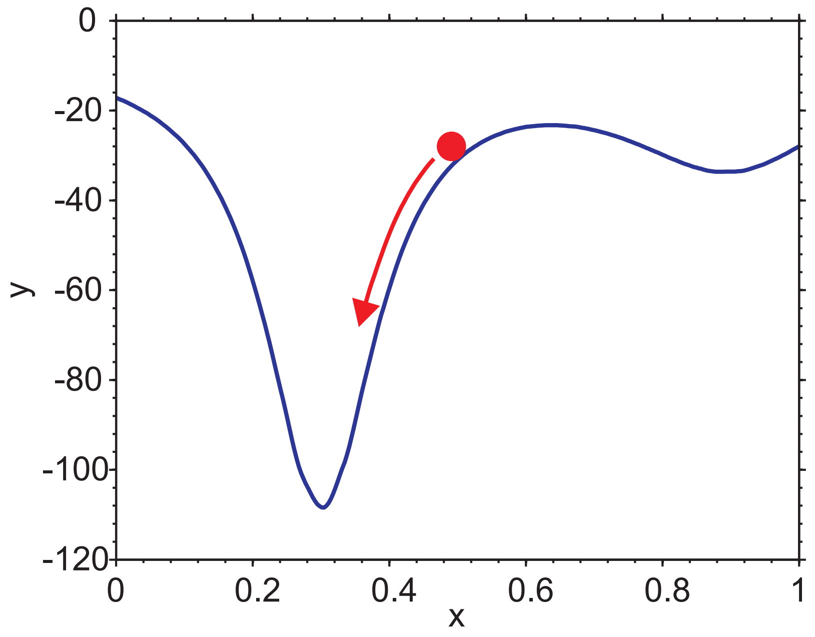


Figure 9.3: *From its starting point at x = 0.5, the ball rolls down the hill until it stops in the deepest part of the valley, which is our minimum at x = 0.3004*

The `fminsearch` function performs this task iteratively from the provided starting point, if increasing x produces a decrease in y, then a small positive change is made to the estimate of x. Alternatively, if decreasing x results in decreasing y then a small negative change is made to our estimate of x. In this way the algorithm follows the value of the function downhill, hence the name "gradient descent". This approach is simple but it does have its limitations, lets look at what happens if we try to find the minimum of the function from a starting point of $x = 1.3$.

```
>> x0 = fminsearch(@(x) myfunction(x), 1.3)

x0 =

0.8927
```

We have obtained a different result! However if we look at the shape of the function it is not difficult to see why this is the case. The gradient descent method will always move "downhill" so from our starting point at $x = 1.3$ it heads towards the small valley at $x = 0.8927$, but then gets stuck. Once at the bottom of this valley y is increasing when x is either increased or decreased, so the algorithm would effectively have to travel "uphill" in order to reach the minimum at $x = 0.3004$, which is not allowed.

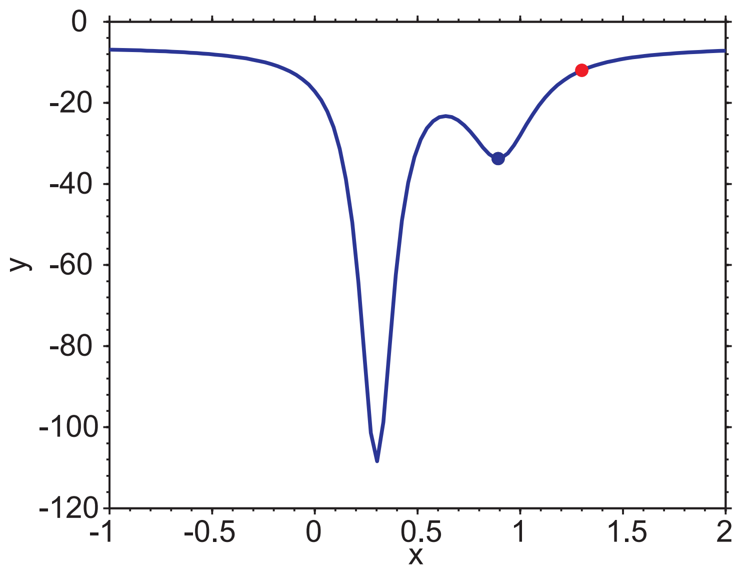


Figure 9.4: *Minimization of equ. 9.1, from the starting point x = 1.3. The starting point is shown as a red circle, the finishing point as a blue circle.*

Here we must introduce the concept of *global* vs. *local* minima.

- *global* minimum: the point at which the function achieves its least value.
- *local* minimum: a point in the function, f, where $f(x) \leq f(y)$ for all points y in a neighborhood of x.

With this in mind your initial guess becomes very important because it will control wether you obtain the global or the local minimum. The function we used in equ 9.1 is very simple, we can plot it and see that a local minimum exists, this may not be possible for more complicated functions. For the kind of problems we are working with we must always be aware that the algorithm might be returning a local rather than global minimum. One simple way to investigate for the presence of local minima in your function is to run it lots of times with different starting values and see if different answers are returned.

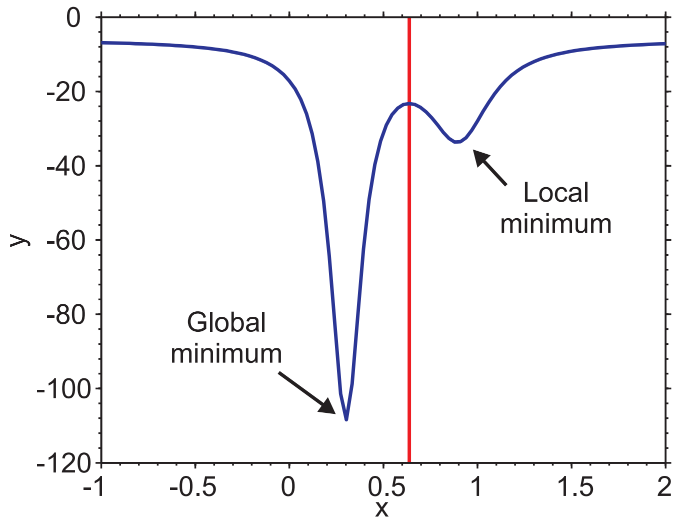


Figure 9.5: *The function defined by equ. 9.1 has both a global and a local minimum. If we use the gradient descent method an initial guess to the right of the red line will return only the local minimum, whilst starting from the left of the line will return the global minimum.*

Some algorithms have been proposed which are claimed to always find the global minimum of a function, but they only work for a very specific set of problems. An example of a more complex algorithm in MATLAB is the function `fminbnd`, which (in theory) will return the location of the deepest minimum of a function within a given range of x. So for example we would use `fminbnd` to find the location of the deepest minimum of equ. 9.1 within the range $-1 \leq x \leq 2$. Notice that we don't supply the function with an initial guess, only the bounds of the neighborhood of x that it should investigate (in this case the last 2 inputs of the function).

```
>> x0 = fminbnd(@(x) myfunction(x),-1,2)

x0 =

0.3004
```

The algorithm correctly identifies that the minimum at $x = 0.3004$, is deeper than the one at $x = 0.8927$, however there is still no guarantee that this is the true global minimum, which could lie outside the range $-1 \leq x \leq 2$. (In my person experience `fminbnd` does not always return the minimum within a given range, so be careful when using it).

Another interesting question is to consider finding the maxima of the function rather than the minima. In fact this is very simple to deal with, we would multiply equ. 9.1 by -1 to turn the maxima into minima and then proceed with the minimization procedure as before.

9.2 Minimization in higher dimensions

The gradient descent algorithm can be applied to higher dimension problems and still performs in the same way, following the function downhill until it reaches a minimum. Let's look at a more complicated two-dimensional function, which calculates a value z based on inputs x and

y.

$$\begin{aligned} z = f(x,y) =& 3(1-x)^2 \exp(-x^2-(y+1)^2) \\ & -1\left(\frac{x}{5}-x^3-y^5\right)\exp(-x^2-y^2) \\ & -\frac{1}{3}\exp(-(x+1)^2-y^2) \end{aligned} \tag{9.2}$$

This is probably a bit too complicated to write as a single inline function, so again we will write a separate m-function file, but we have to take into account the manner in which `fminsearch` works. The `fminsearch` function only allows one variable to be changed during the minimization, therefore we must input x and y together as a single variable. The easiest way to do this is to combine x and y for the call to the function, but then split them once they are inside the function. Therefore we'll use one input variable xy, the first column of which is the x data and the second column of which is the y data.

```
function z=myfunction2(xy)
%function to calculate equ. 9.2
%to make things easier we'll split the calculation into 3 parts

x=xy(:,1); y=xy(:,2) split the data into its x and y components
A=3.*(1-x).^2.*exp(-x.^2-(y+1).^2);
B=-10.*(x./5-x.^3-y.^5).*exp(-x.^2-y.^2);
C=-1./3.*exp(-(x+1).^2-y.^2);

z = A + B + C;
```

First we should look at the shape of the function, to act as an input we need matrices representing x,y. To do this we can use the `meshgrid` function that we used previously:

```
>> [x,y]=meshgrid(-3:0.1:3,-3:0.1:3); %generate the (x,y) data
>> z = myfunction2([x(:),y(:)]); % combine x and y, and determine the value
>> surfc(x,y,z) %plot as a surface with contour lines
>> colormap gray %use the grayscale color map
>> xlabel('x') %add a label to the x-axis
>> ylabel('y') %add a label to the y-axis
>> zlabel('z') %add a label to the z-axis
```

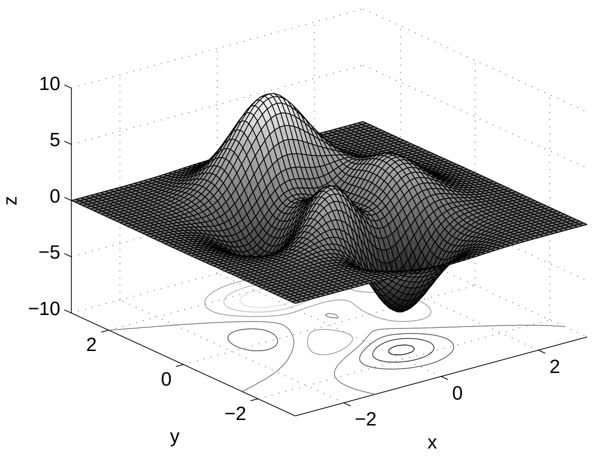


Figure 9.6: *The two-dimensional function defined by equ. 9.2. Use the Rotate3D option to examine the topography, you should see that there are three minima; two local and one global.*

We can locate the minima in equ. 9.2 using `fminsearch` in the same manner as we did for the one-dimensional case. The only difference is that because we are now working with x and y coordinates we need to supply a first guess for each one (entered into function as a single array). Lets start with an initial guess of *(1.1,-0.6)*:

```
>> hold on
>> xy0 = fminsearch(@(xy) myfunction2(xy),[1.1,-0.6])
>> plot3(1.1,-0.6,myfunction2([1.1 -0.6]),'or') %plot the starting point
>> plot3(xy0(1),xy0(2),myfunction2(xy0),'ob') %plot the finishing point
```

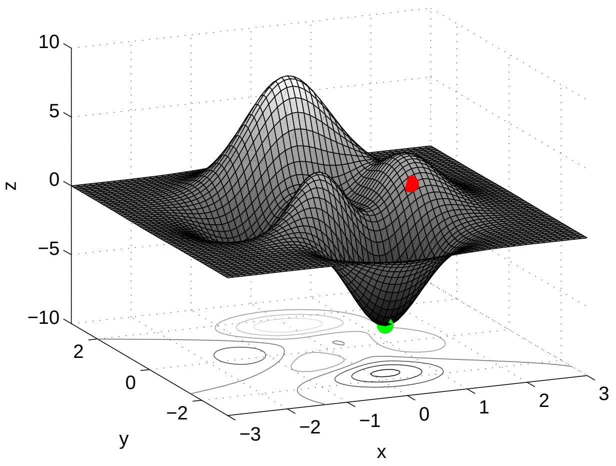


Figure 9.7: *Minimization of equ. 9.2, using fminsearch from a starting point of (1.1,-0.6). The starting point is shown as a red circle, the finishing point as a green circle.*

As with the one-dimensional case, `fminsearch` will work downhill towards a minimum, which may not necessarily be the global minimum.

The one and two-dimensional cases that we have examined so far have been very simply where you could minimize the function quite accurately by eye. Now we will look at more complicated situations where the power of optimization will hopefully become evident.

9.3 Constrained Optimization

In many cases it is desirable to force our minimization problem to fit certain constraints. For example, we may fit a curve to data and know from theory that it should be constrained to have a certain form. Lets start with a curve fitting problem of this form, imagine we are studying a data set with the which follows the cubic relationship:

$$f(x) = c_1x^3 + c_2x^2 + c_3x + c_4 \tag{9.3}$$

Now we'll simulate some experimental data that follows this relationship. To do this we will calculate the values of the function for different values of x and add some random numbers to the results to simulate experimental noise.

```
>> coef = [1 -2 1 -1]; %coefficients of the cubic curve
>> x = linspace(-2,4,100); %an array of 100 x values between -2 and 4
>> y = coef(1).*x.^3 + coef(2).*x.^2 + coef(3).*x + coef(4); %calculate the curve

>> y = y + randn(size(y)); % add random numbers to the curve to act as noise
>> plot(x,y,'.b') %plot the synthetic data as blue points
```

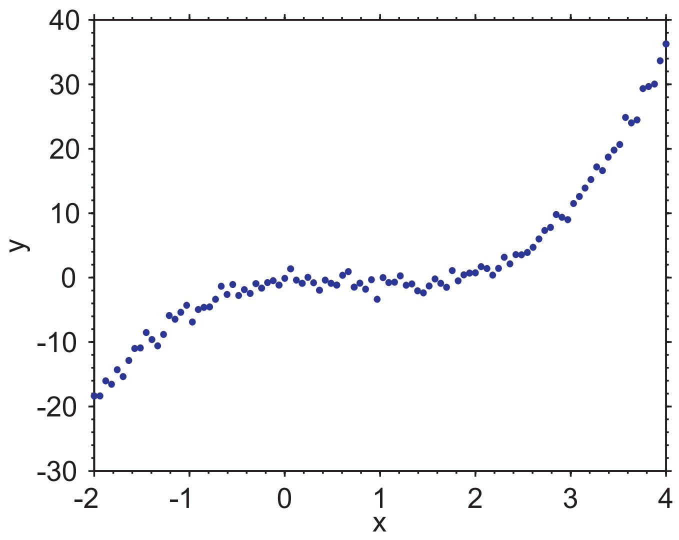


Figure 9.8: *Simulated experimental data following the relationship given in equ. 9.3. Your data may not look exactly the same because the random numbers used to represent the noise will be different.*

We can use minimize the "sum of squared residuals" to find the best fit to the data. In this case we will fit a cubic polynomial to the data, but of course we won't get a perfect fit because we have added a noise component. First we need to calculate the value of the line for a given set of coefficients (which we will store in the vector, c) and then determine the difference to the data, i.e. determine the residuals.

```
function ssr = objfun(c,xdata,ydata)
%objective function to calculate residuals on a cubic fit
%calculate the line given the coefficients in c
yhat=c(1).*xdata.^3 + c(2).*xdata.^2 + c(3).*xdata + c(4);
%calculate the sum of the squared residuals by comparing to ydata
ssr = sum((ydata-yhat).^2);
```

The optimal fit to the data is found when we minimize the sum of the squared residuals. We can do this using `fminsearch`, with a starting guess of c of all zeros. When we have the optimized version of c we can calculate and plot the best fit line for our different values of x.

```
>> c = fminsearch(@(c) objfun(c,x,y),[0 0 0 0]) %minimize the residuals
>> y0 = c(1).*x.^3 + c(2).*x.^2 + c(3).*x + c(4); %the fitted cubic curve
>> hold on
>> plot(x,y0,'r','linewidth',1) %plot the fitted curve as a red line
```

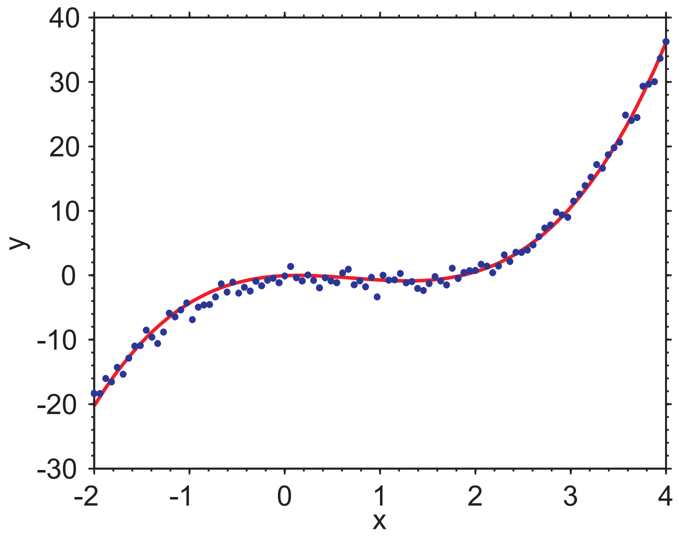


Figure 9.9: *Optimized cubic fit (red line) to the experimental data (blue dots).*

The fit seems to be okay, but imagine the situation where theory tells us the fitted curve has to obey certain constraints. The function `fminsearch` can only perform unconstrained minimization, so instead we must use the function `fmincon`, which has addition inputs which allow you to add problem constraints. The inputs are quite complex, but they allow us to define linear inequalities, linear equalities and bounds; they are as follows:

```
c = fmincon(function,c0,A,b,Aeq,beq,lb,ub)
```

$$\text{minimize f(c) such that: } \begin{cases} A \cdot c \leq b \\ Aeq \cdot c = beq \\ lb \leq c \leq ub \end{cases} \tag{9.4}$$

So what does all of this mean? The constraints allow us to find a solution to the minimization problem which meet certain requirements. Lets look at the statements one by one, in all cases the final solution of the minimization is represented by c:

- $A \cdot c \leq b$, we can define a set of values in A that when matrix multiplied by c must be less than or equal to the defined value b.
- $Aeq \cdot c \leq beq$, we can define a set of values in A that when matrix multiplied by c must be equal to the defined value b.
- $lb \leq c \leq ub$, all the values in c must be greater than or equal to the lower bound, lb, and less than or equal to the upper bound, ub.

9.3.1 Linear Equalities

We'll start by enforcing a constraint that the curve must pass through the point (-2,-19). To do this we must use the linear equalities option, which can be written in the form:

$$A = [-2^3, -2^2, -2, 1]$$

$$A \cdot c = -19$$

In MATLAB notation we can set the constraints and call the function in the following way:

```
>> xc = -2 %define the constrained x value
>> yc = -19 %define the constrained y value
>> Aeq=xc.^[3,2,1,0]; %left-hand side of the constraint
>> beq=yc; %right-hand side of the constraint
>> c=fmincon(@(c) objfun(c,x,y),[0 0 0 0],[],[],Aeq,beq,[],[]); %call fmincon
>> y0 = c(1).*x.^3 + c(2).*x.^2 + c(3).*x + c(4); %the fitted cubic curve
>> plot(x,y0,'k','linewidth',1) %plot the fitted curve as a black line
```

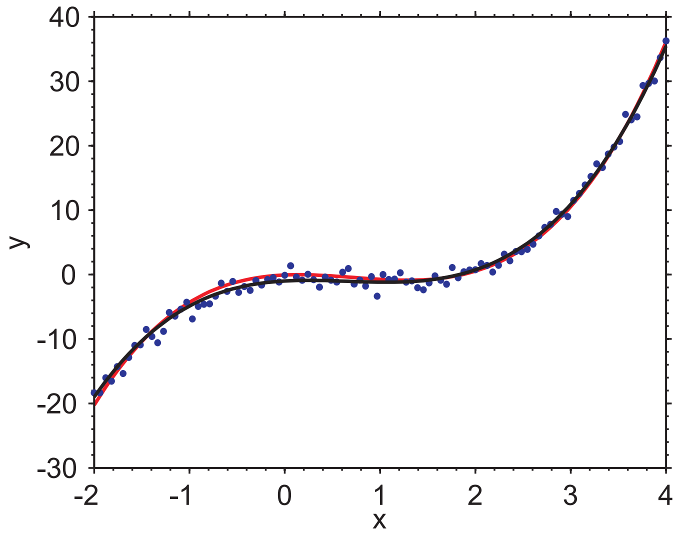


Figure 9.10: *Optimized cubic fit (black line) to the experimental data (blue dots) constrained to pass through the point (-2,-19).*

We can look at the returned coefficients in `c`, and check that `Aeq · c` does in fact equal `beq`.

```
>> c'%display the coefficients as a column

ans =
1.0281
-2.0414
0.8203
-0.9689

>> Aeq*c'%check the equality constraint

ans =
-19
```

To add more equality constraints is simple, we just need to combine them into a matrix with one for each constraint. We have our first constraint that the fitted line must pass through the point (-2,-19), now lets add another constraint to also force the line through (0,-1):

```
>> Aeq(1,:)=(-2).^[3,2,1,0]; %left-hand side of the 1st constraint
>> beq(1,1)=-19; %right-hand side of the 1st constraint
>> Aeq(2,:)=0.^[3,2,1,0]; %left-hand side of the 2nd constraint
>> beq(2,1)=-1; %right-hand side of the 2nd constraint
>> c=fmincon(@(c) objfun(c,x,y),[0 0 0 0],[],[],Aeq,beq,[],[]); %call fmincon
>> y2 = c(1).*x.^3 + c(2).*x.^2 + c(3).*x + c(4); %the fitted cubic curve
>> plot(x,y2,'g','linewidth',1) %plot the fitted curve as a green line
>> Aeq*c'%check the inequality constraint

ans =
-19.0000
-1.0000
```

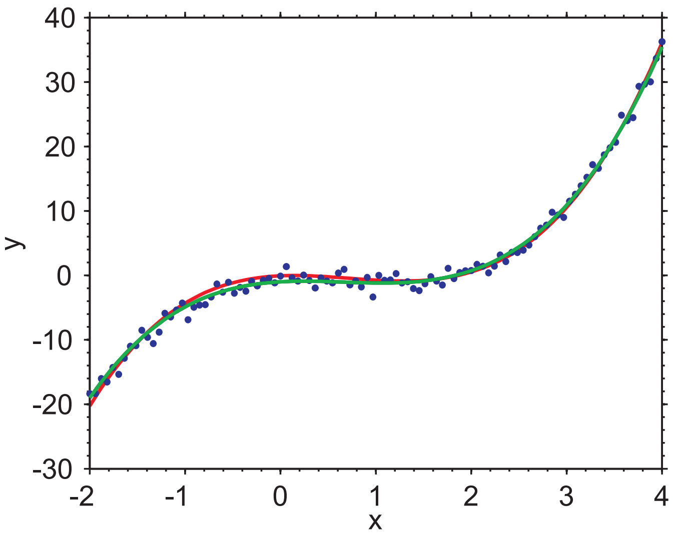


Figure 9.11: *Optimized cubic fit (green line) to the experimental data (blue dots) constrained to pass through the points (-2,-19) and (0,-1).*

9.3.2 Linear inequalities

Now lets try a linear inequality, forcing the fitted curve to have a gradient greater than or equal to 5 when $x = 2$. First we need an expression for the gradient of the cubic polynomial, a bit of simple calculus gives us:

$$f'(x) = 3c_1x^2 + 2c_2x + c_3 \tag{9.5}$$

We can now write the constrain on the gradient based on equ. 9.5:

```
>> x1=2
>> y1=5
>> A=[3.*x1.^2, 2.*x1, 1, 0]
>> b=y1
```

Before we call the function it is important to note that our constraint demands that the gradient is greater than or equal to a given value, whilst the function uses linear inequalities of the form less than or equal to. We can get around this problem very simply by using $-A$ and $-b$ as the constraints, also keeping our previous linear equality the function call looks like:

```
>> c=fmincon(@(c) objfun(c,x,y),[0 0 0 0],-A,-b,Aeq,beq,[],[]);
```

We can check that the inequality constraint is met and then plot the resulting line:

```
>> A*c'%check the inequality constraint

ans =

5

>> y1 = c(1).*x.^3 + c(2).*x.^2 + c(3).*x + c(4); %the fitted cubic curve
>> plot(x,y1,'b','linewidth',1) %plot the fitted curve as a blue line
```

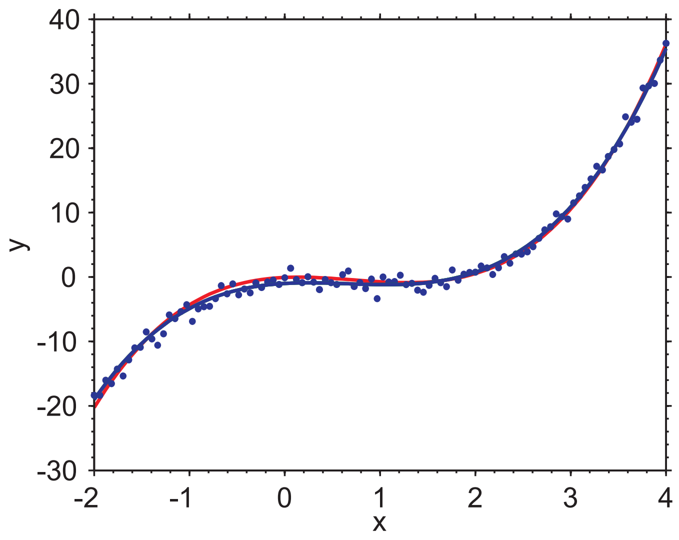


Figure 9.12: *Optimized cubic fit (blue line) to the experimental data (blue dots) constrained to pass through the points (-2,-19) and (0,-1) and to have a gradient greater than or equal to 5 at* $x = 2$.

9.4 Exercise: The Geiger counter

Often a Geiger counter will be used to measure the radioactivity of rocks. One problem with simple Geiger counters is that at high activity (high count rates) they will underestimate the true activity. This is because after a γ-ray enters the detector the system is "dead" for a short period of time during which it cannot measure. If another γ-ray enters the system during this dead-time it will not be counted and thus the observed count rate will be less than the true count rate. This situation is made worse because not only are the γ-rays which enter during the dead-time not counted, they do extend the dead-time.

Figure 9.13: *How can we find the true count rate of a Geiger counter which suffers from dead-time.*

The relationship between the observed count rate (N_{obs}) and the true count rate (N_{true}) is an exponential law equation:

$$N_{obs} = N_{true}e^{-N_{true}\tau}$$

where τ is the dead-time per pulse (20 x 10^{-6} seconds in old instruments). Note this is a transcendental equation which means that it cannot be rewritten in the form N_{true} = ... In this exercise you will use `fminsearch` to determine the value of N_{true} when N_{obs} = 8000 *cps* and τ = 20 x 10^{-6} s.

Steps to completing the exercise

Don't forget that we have examined this problem before when we discussed writing functions. In the previous case we found our best estimate by finding the value of N_{true} which produced the minimum squared difference of the count rate equation with the observed count rate.

- Find an expression which you can minimize to find the best estimate of N_{true}.
- Write this expression as an `inline` function.
- Minimize the function using `fminsearch` and an initial guess of N_{true}.

Functions you may find useful

- `exp` - Exponential.
- `inline` - Construct INLINE object.
- `fminsearch` - Multidimensional unconstrained nonlinear minimization.

Chapter 10

Image Analysis

Image analysis is a complex field which provides a number of methods with which to process image data. Most images are nothing more than matrices of pixels, therefore they can be processed extremely efficiently in MATLAB. An “Image Processing Toolbox” is available from The MathWorks which contains a large number of M-functions that provide state of the art analysis methods (which tend to be pretty mathematically intensive). Such a toolbox is however unnecessary for simple processing and many of the basic tasks in image processing are nothing more that simple matrix manipulations. Here we are going to look at some simple examples which consider processes such as cropping, rotation and color equalization, if you would like more information then you can look at the Image Processing Toolbox or books such as Digital Image Processing Using MATLAB by Gonzalez and Eddins.

Before we get started with the examples it is important to consider briefly what form image data takes (ultimately this will define how we have to process it). We will be working with raster graphics which are nothing more that a collection of colored pixels on a regular grid. Each pixel is given a defined color which is stored numerically as a combination of different parameters, for example a defined mixture of red, green and blue. Considering the example of a red, green and blue mixture, the so-called RGB color model, for any given pixel we need to provide a representation of the intensity of the red, green and blue components of its colour. This means for each pixel we will need to define three numbers, one for each of the color “channels” and there are a number of ways we can do this:

- From 0 to 1, with any fractional value in between. This representation is used in theoretical analyses, and in systems that use floating-point representations.
- Each color component value can also be written as a percentage, from 0% to 100%.
- In computing, the component values are often stored as integer numbers in the range 0 to 255, the range that a single 8-bit byte can offer (by encoding 256 distinct values).
- High-end digital image equipment can deal with the integer range 0 to 65,535 for each primary color, by employing 16-bit words instead of 8-bit bytes.

In most cases MATLAB employs the 0 to 255 system to define RGB triplets, which allows the information to be stored more efficiently as unsigned 8-bit integers rather than double precision numbers which require 64-bits (this means for some operations we will have to convert the data to double precision before we can proceed). A few examples of colors in the RGB model are:

- `[255,0,0]` → red
- `[0,255,0]` → green
- `[0,0,255]` → blue
- `[255,255,255]` → white
- `[0,0,0]` → black
- `[127,255,212]` → aquamarine

Later when we start loading images into MATLAB we will see how the information on the different colour channels is stored. The other important colour model for us to consider is "grayscale" which only carries intensity information, representing a mixture of black and white. As with the RGB model, grayscale normally ranges between 0 (black) and 1 or 255 (white), therefore for each pixel we only need to store one unsigned 8-bit integer.

10.1 MATLAB and images

A number of M-functions are available in the standard release of MATLAB with which to read, display and save images. In this first example we will recover information about an image, then load it and look how it is structured before finally displaying it. Over the following examples there will be a number of repeated commands, so it would be a good idea to produce a script function from which you can easily recall certain tasks. We will clear the memory and close all existing figures before using the function `imfinfo` to obtain information on the properties of the image file *thin_section1.jpg*.

```
>> clear all, close all
>> info=imfinfo('thin_section1.jpg')

info =

Filename:  'C:\Documents and Settings\thin_section1.jpg'
FileModDate:  '08-Oct-2009 16:45:58'
FileSize:  413876
Format:  'jpg'
FormatVersion:  ''
Width:  640
Height:  449
BitDepth:  24
ColorType:  'truecolor'
FormatSignature:  ''
```

```
NumberOfSamples:  3
CodingMethod:  'Huffman'
CodingProcess:  'Sequential'
Comment:  {}
```

As you can see this provides us with detailed information concerning the properties of the image. We can load the image into the memory using the `imread` function and we will store the numeric data representing the pixel colors in an array called `I`. We will then look at the `size` of `I` to understand how the data is organized:

```
>> I = imread('thin_section1.jpg');
>> size(I)
ans =
449 640 3
```

The size information reveals that we have a three-dimensional matrix, in the third dimension we have 3 layers of data. You can think of this structure as a collection of 3 two-dimensional matrices which are 449 by 640 lying on top of each other. The different layers represent the different color channels in the sequence; red (first layer), green (second layer) and blue (third layer). So if we wanted to look at the RGB values for the first pixel in the image (top lefthand corner) we can simply call it by its index:

```
>> I(1,1,:)

ans(:,:,1) =
2

ans(:,:,2) =
2

ans(:,:,3) =
12
```

which means it has an RGB triplet of `[2,2,12]`. Finally, we are going to display the image using the function `imagesc`, once this is plotted we need to adjust the aspect ratio of the image to ensure the pixels are square (by default MATLAB plots with an xy aspect ratio correspond to the golden section so in the case of images we need to adjust this ratio). As a last step we will remove the plot axes which are included in the figure by default:

```
>> imagesc(I) %diplay image
>> set(gca,'dataaspectratio',[1 1 1]) %equal xy scaling
>> set(gca,'visible','off') %remove axes
```

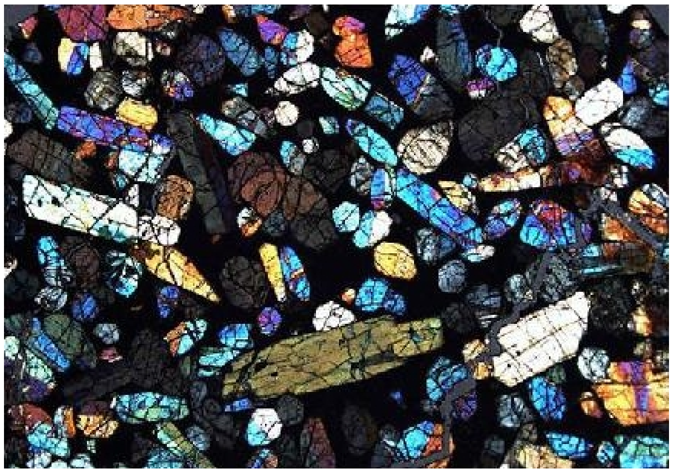


Figure 10.1: *The RGB image stored in the file thin_section1.jpg.*

10.2 Intensity information and Colormaps

One important operation is to be able to convert an image from the RGB color model in the gray scale color model. This is actually surprisingly easy, all it requires is for us to convert the RGB triplets into intensity information using a bit of book-keeping (making sure all the information is stored in the right places) and a matrix multiplication with some predefined transfer coefficients. We'll plot the final result and then also look at how we can change the appearance using MATLAB's various colormaps. To start with we'll clear the memory and reload the original image.

```
>> clear all, close all
>> I = imread('thin_section1.jpg'); %load image as 3D RGB array
```

We're now going to separate the three-dimensional matrix into a collection of three two-dimensional matrices, one for each color channel. This means extracting each layer and using the `squeeze` function to reduce it from a 3D 449 by 640 by 1 matrix to a 2D 449 by 640 matrix (as you might of guessed the `squeeze` function removes singleton dimensions).

```
>> R=squeeze(I(:,:,1)); %obtain a 2D copy of red channel information
>> G=squeeze(I(:,:,2)); %obtain a 2D copy of green channel information
>> B=squeeze(I(:,:,3)); %obtain a 2D copy of blue channel information
```

We'll now recombine the color information in the form of 3 vectors; red, green and blue which each contains 287360 entries (i.e. the product of 449 and 640) and convert the numbers to double precision.

```
>> RGB=[R(:),G(:),B(:)]; %form RGB into a 3 column matrix
>> RGB=double(RGB); %convert to double precision
```

In order to derive the intensity values we now perform a matrix multiplication between the matrix RGB and some predefined coefficients and then check that no values lie outside the allowable 0 to 255 range. Once this done we simply need to reorganize the pixels into their original positions thus restoring the structure of the image.

```
>> coef=[0.2989; 0.5870; 0.1140] %RGB to intensity transform
>> GS=RGB*coef; %(matrix) multiply RGB and transform coefficients
>> GS(GS<0)=0; %round up values below 0
>> GS(GS>255)=255; %round down values above 255

>> [r,c]=size(I(:,:,1)) %rows and columns of original image
>> GS=reshape(GS,[r c]); %change from a vector to a matrix
```

We now have a two-dimensional matrix which has the same size and shape as the original image and contains individual intensity values for each of the pixels. We can plot this intensity information using the `imagesc` function and then change its appearance using various colormaps (a grayscale color model is the most traditional for this kind of data).

```
>> imagesc(GS) %diplay intensities with default jet colormap
>> set(gca,'dataaspectratio',[1 1 1]) %set equal x y scaling
>> set(gca,'visible','off') %remove axes
>> colormap gray(256) %change colormap to "gray" with 256 levels
>> colormap cool(256) %change colormap to "cool" with 256 levels
>> colormap summer(256) %change colormap to "summer" with 256 levels
```

Finally, if we can output the image using the function `imwrite` which has additional arguments to control the type of image, level of compression etc (see the help entry of `imwrite` for more information).

```
>> imwrite(GS,summer(256),'test.jpg','quality',90) %output image as jpg
```

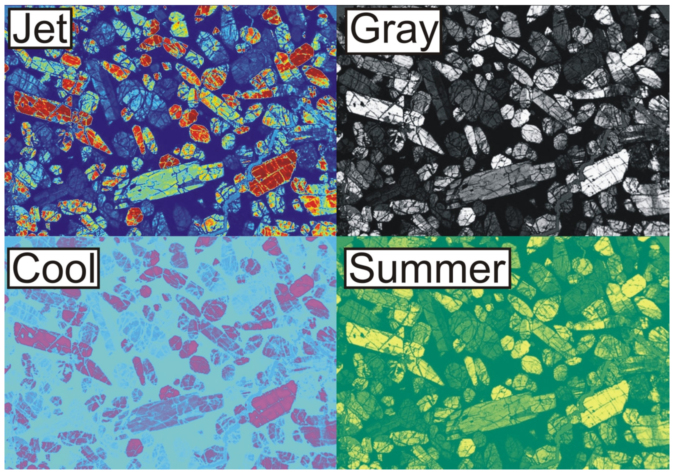


Figure 10.2: *The intensity image displayed with a variety of different colormaps*

10.3 Cropping an image

Cropping images is a simple task of identifying the outmost pixels of the region of interest and producing a copy of the image data up to these boundaries. In the following example we study how to crop an intensity image and then extend the idea to RGB images. We'll also define the limits of the region of interest by interacting with the image directly using the `ginput` function. Notice the must of the first part of this exercise is a repeat of previous exercises (loading the image, converting it to intensity values etc), but it is still included for completeness.

```
>> clear all, close all
>> I = imread('thin_section1.jpg'); %load image as 3D RGB array

>> R=squeeze(I(:,:,1)); %obtain copy of red channel information
>> G=squeeze(I(:,:,2)); %obtain copy of green channel information
>> B=squeeze(I(:,:,3)); %obtain copy of blue channel information

>> RGB=[R(:),G(:),B(:)]; %form RGB into a 3 column matrix
>> RGB=double(RGB); %convert to double precision

>> coef=[0.2989; 0.5870; 0.1140] %RGB to intensity transform
>> GS=RGB*coef; %(matrix) multiply RGB and transform coefficiants
>> GS(GS<0)=0; %round up values below 0
>> GS(GS>255)=255; %round down values above 255

>> [r,c]=size(I(:,:,1)) %rows and columns of original image
>> GS=reshape(GS,[r c]); %change from a vector to a matrix

>> imagesc(GS) %diplay intensities with default jet colormap
>> set(gca,'dataaspectratio',[1 1 1]) %set equal x y scaling
>> set(gca,'visible','off') %remove axes
>> colormap gray(256) %change colormap to "gray" with 256 levels
```

At this stage we have loaded the image, converted it to magnitude data and displayed the results. Next we'll use the `ginput` function to display the select the region on interest. When used in the form `ginput(n)` the function will bring the active image to the front and using a set of cross-hairs allow use to define `n` positions with a click of the mouse. The `x` and `y` positions of these locations will be stored in the output variables. In this case we need to define two positions which are the opposite corners of the region of interest. Because we will employ the `x` and `y` values as addresses they must be rounded to the nearest integer (using the `round` function) and then we will make a copy of the intensity data between these limits. Notice the use of the `min` and `max` functions when defining the addresses, this is necessary because we don't know in which order the user selected the corner points.

```
>> [xlim,ylim]=ginput(2) %interact with image to define 2 corner points
>> xlim=round(xlim); %round x to nearest integer (i.e.  to nearest pixel)
>> ylim=round(ylim); %round y to nearest integer (i.e.  to nearest pixel)
>> GSc=GS(min(ylim):max(ylim),min(xlim):max(xlim)); %select rows and columns
```

The cropped version of GS now resides in the variable GSc, which can be plotted in the same manner as other intensity images:

```
>> figure %create new figure window
>> imagesc(GSc) %diplay cropped image
>> set(gca,'dataaspectratio',[1 1 1]) %set equal x y scaling
>> set(gca,'visible','off') %remove axes
>> colormap gray(256) %change colormap to "gray" with 256 levels
```

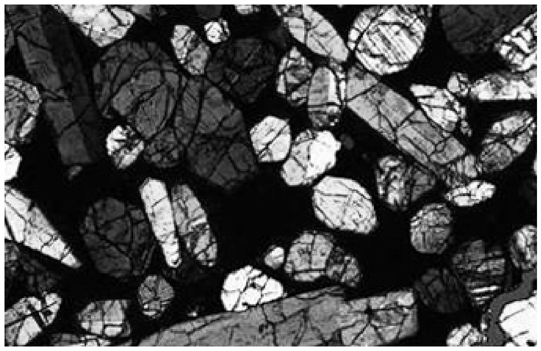


Figure 10.3: *Cropped intensity image displayed with a grayscale colormap*

10.4 Cropping a colour image

The ideas behind cropping colour images are exactly the same as intensity images. The key difference is that now we are working with a three rather than two dimensional matrix and we must crop each of the red, green and blue layers appropriately. The first part of this is example follows the ideas introduced above, load the image, display it and select the crop points using the `ginput` function.

```
>> clear all, close all

>> I = imread('thin_section1.jpg'); %load image as 3D RGB array
>> imagesc(I) %display image
>> set(gca,'dataaspectratio',[1 1 1]) %set equal x y scaling
>> set(gca,'visible','off') %remove axes

>> [xlim,ylim]=ginput(2) %interact with image to define 2 crop points
>> xlim=round(xlim); %round x to nearest integer (i.e.  to nearest pixel)
>> ylim=round(ylim); %round y to nearest integer (i.e.  to nearest pixel)
```

We can now use the limits in `xlim` and `ylim` to act as addresses to crop the image across all three layers.

```
%now select rows and columns of the ROI and all 3 color layers
>> Ic=I(min(ylim):max(ylim),min(xlim):max(xlim),:);
>> figure %create new figure window
>> imagesc(Ic) %diplay cropped RGB image
>> set(gca,'dataaspectratio',[1 1 1]) %set equal x y scaling
>> set(gca,'visible','off') %remove axes
```

If we want to output this image using `imwrite` there are two important differences to the intensity example we looked at earlier. The first is the data must be converted from double precision into unsigned 8-bit integers, this can be achieved with the command:

```
>> Ic=uint8(Ic);
```

Secondly when we give the `imwrite` command we do not need to define a colormap because we are providing a collection of RGB values (i.e. the colors are defined by the data itself).

```
>> imwrite(Ic,'test.jpg','quality',90)
```

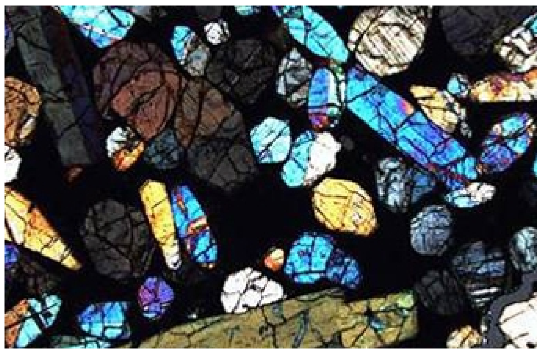


Figure 10.4: *Cropped RGB image*

10.5 Resizing and Aspect ratios

In this example we will work with an intensity image, but exactly the same procedure could be applied to an RGB image by processing the individual red, green and blue layers separate two-dimensional matrices and then recombining them into a three-dimensional matrix. To begin with we will load the image and convert it into intensity data (using the same commands as above):

```
>> clear all, close all
>> I = imread('thin_section1.jpg'); %load image as 3D RGB array

>> R=squeeze(I(:,:,1)); %obtain copy of red channel information
>> G=squeeze(I(:,:,2)); %obtain copy of green channel information
>> B=squeeze(I(:,:,3)); %obtain copy of blue channel information

>> RGB=[R(:),G(:),B(:)]; %form RGB into a 3 column matrix
>> RGB=double(RGB); %convert to double precision
>> coef=[0.2989; 0.5870; 0.1140] %RGB to intensity transform
>> GS=RGB*coef; %(matrix) multiply RGB and transform coefficiants
>> GS(GS<0)=0; %round up values below 0
>> GS(GS>255)=255; %round down values above 255

>> [r,c]=size(I(:,:,1)) %rows and columns of original image
>> GS=reshape(GS,[r c]); %change from a vector to a matrix

>> imagesc(GS) %diplay intensities with default jet colormap
>> set(gca,'dataaspectratio',[1 1 1]) %set equal x y scaling
>> set(gca,'visible','off') %remove axes
>> colormap gray(256) %change colormap to "gray" with 256 levels
```

To consider the problems of resizing and adjusting aspect ratios it is important to think about how the image data is organized. We have a series of square pixels positioned on a regularly spaced grid, which in this specific case has 449 rows and 640 columns. We can generate appropriately spatial grids representing the positions using the `meshgrid` function and assuming the upper-left pixel has a position of `[0,0]` and the lower-right pixel has a position of `[1,1]`.

```
% make x and y grids to represent pixel positions
>> [x,y]=meshgrid(linspace(0,1,size(GS,2)),linspace(0,1,size(GS,1)));
```

The next step is to define factors in the by which to contract or expand the figure in both the x and y directions (where values greater than 1 represent expansion and those less than 1 represent contraction).

```
>> xf=2.0 %reduce resolution in x axis by 50%
>> yf=0.5 %increase resolution in x axis by 275%
```

Now we'll make two new spatial grids (`xn` and `yn`) which again range between `[0,0]` and `[1,1]` but have the increments between points in the x and y directions adjusted by the predefined factors. This means were we want to expand the image we will include more points and alternatively where we want to contract the image we will include less points. Once these grids are produced we simply need to interpolate the image data from the original `x` and `y` grids onto the new `xn` and `yn` grids.

```
%change resolution in x and y directions by given factors
>> [xn,yn]=meshgrid(linspace(0,1,size(GS,2).*xf),linspace(0,1,size(GS,1).*yf));
>> GSs=griddata(x,y,GS,xn,yn); %interpolate image on to new grid points
```

Because the spatial information of the image is inherent to the structure of the data (as mentioned above, pixels are square and assumed to be equally spaced in the x and y directions) we have now expanded and compressed the image by adding and removing pixels, respectively. The last step is to display the rescaled image.

```
>> figure %create new figure window
>> imagesc(GSs) %diplay cropped image
>> set(gca,'dataaspectratio',[1 1 1]) %set equal x y scaling
>> set(gca,'visible','off') %remove axes
>> colormap gray(256) %change colormap to "gray" with 256 levels
```

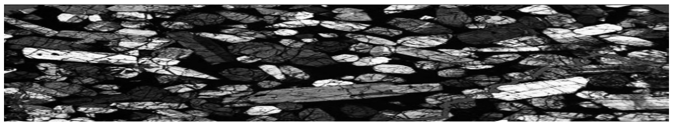


Figure 10.5: *Intensity image expanded to 200% in the x direction and shrunk to 50% in the y direction*

10.6 Image Rotation

Rotating an image follows the same ideas as the previous example, we manipulate the original positions of the pixels and then interpolate onto a regular grid. In this example we will look at an RGB image, so the 3 layers will need to be treated separately. The first task is to load the image and display it.

```
>> clear all, close all
>> I = imread('thin_section1.jpg'); %load image as 3D RGB array

>> imagesc(I) %diplay image
>> set(gca,'dataaspectratio',[1 1 1]) %set equal x y scaling
>> set(gca,'visible','off') %remove axes
```

Next we will define the angle (in radians) we would like to rotate the image through and use this to define a two-dimensional rotation matrix of the form:

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

In code this looks like:

```
>> phi=45./180.*pi; %choose rotation angle and convert to radians
>> T=[cos(phi) -sin(phi); sin(phi) cos(phi)]; %2D rotation matrix
```

To perform the rotation we need to define a set of pseudo x,y positions for the pixels. An important point to consider here is that the rotation matrix we have constructed will rotate about the point $(0, 0)$, therefore if we want to rotate the image about its center our grids of positions must range between an upper-left position of `[-0.5,-0.5]` and a lower-right position of `[0.5,0.5]`.

```
>> [r,c,l]=size(I) %image size (rows, columns, layers)
```

```
%generate pseudo x,y positions for each of the pixels
>> [x,y]=meshgrid(linspace(-0.5,0.5,c),linspace(-0.5,0.5,r));
```

Now we can form the x and y pixel positions into column vectors and combine them into a matrix. To rotate the positions we simply perform a matrix multiplication with the rotation matrix.

```
>> xy=[x(:),y(:)]; %set positions as vectors and combine into a matrix
>> xyT=xy*T; %(matrix) multiply position and rotation matrix
>> xT=xyT(:,1) %rotated x positions
>> yT=xyT(:,2) %rotated y positions
```

We now have a rotated position associated with each of the pixels in the original image, this means that we can take the values for the red, green and blue layers and interpolate them from the rotated coordinates back onto regular coordinate grids which span the range of the rotated coordinates, thus completing the rotation.

```
>> R=squeeze(I(:,:,1)); %obtain copy of red channel information
>> G=squeeze(I(:,:,2)); %obtain copy of green channel information
>> B=squeeze(I(:,:,3)); %obtain copy of blue channel information

>> RGB=[R(:),G(:),B(:)]; %form RGB into a 3 column matrix
>> RGB=double(RGB); %convert to double precision

>> [x,y]=meshgrid(linspace(min(xT),max(xT),c),linspace(min(yT),max(yT),r));
>> Rr=griddata(xT,yT,RGB(:,1),x,y); %interpolate red on regular grid
>> Gr=griddata(xT,yT,RGB(:,2),x,y); %interpolate green on regular grid
>> Br=griddata(xT,yT,RGB(:,3),x,y); %interpolate blue on regular grid
```

We now need to need to reorganize the data into a three-dimensional matrix and the convert it into unsigned 8-bit integers:

```
%create a copy of I into which the rotated data will be placed
>> Ir=double(I);
>> Ir(:,:,1)=Rr;
>> Ir(:,:,2)=Gr;
>> Ir(:,:,3)=Br;
>> Ir=uint8(Ir) %convert data into unsigned 8bit integers
```

Finally we can display the image, note the now empty black areas where no image information exists.

```
>> figure %create new figure window
>> imagesc(Ir) %diplay image
>> set(gca,'dataaspectratio',[1 1 1]) %set equal x y scaling
>> set(gca,'visible','off') %remove axes
```

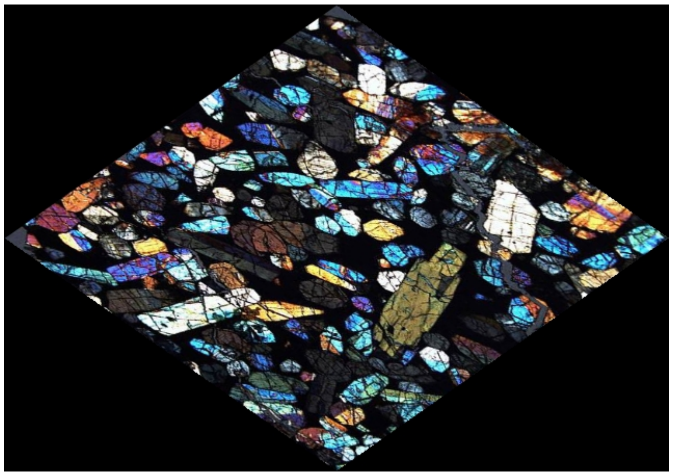


Figure 10.6: *RGB image rotated by 45°*

10.7 Histogram Equalization

Histogram equalization provides a method with which to adjust (enhance) the contrast of an image. This is achieved by flattening and stretching the distribution of pixel intensities across the range 0 to 255. In the following example we will load our standard image, convert it to intensities and produce a histogram of the resulting values.

```
>> clear all, close all

>> I = imread('thin_section1.jpg'); %load image as 3D RGB array

>> R=squeeze(I(:,:,1)); %obtain copy of red channel information
>> G=squeeze(I(:,:,2)); %obtain copy of green channel information
>> B=squeeze(I(:,:,3)); %obtain copy of blue channel information

>> RGB=[R(:),G(:),B(:)]; %form RGB into a 3 column matrix
>> RGB=double(RGB); %convert to double precision

>> coef=[0.2989; 0.5870; 0.1140] %RGB to intensity transform
>> GS=RGB*coef; %(matrix) multiply RGB and transform coefficiants
>> GS(GS<0)=0; %round up values below 0
>> GS(GS>255)=255; %round down values above 255

>> [r,c]=size(I(:,:,1)) %rows and columns of original image
>> GS=reshape(GS,[r c]); %change from a vector to a matrix

>> imagesc(GS) %diplay intensities with default jet colormap
>> set(gca,'dataaspectratio',[1 1 1]) %set equal x y scaling
>> set(gca,'visible','off') %remove axes
>> colormap gray(256) %change colormap to "gray" with 256 levels
```

At this stage we would like to produce a histogram of the intensity values. This can be achieved using the `hist` function to count the number of pixels falling into defined bins and the `bar` function to plot the result.

```
>> N = histc(GS(:),[0:255]); %find number of pixels in each intensity band
>> figure, bar([0:255],N,'histc') %plot histogram
>> xlabel('Intensity [0-255]') %add a x-label
>> ylabel('Number of Pixels') %add a y-label
>> set(gca,'tickdir','out') %make ticks face outwards
```

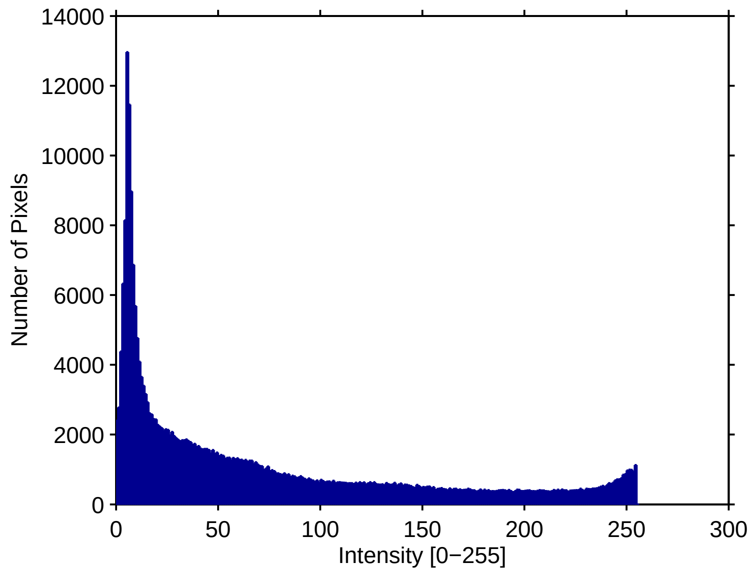


Figure 10.7: *Histogram of pixel intensities*

We can see that the distribution of intensities is strongly skewed towards darker values, in this case we may be able to improve the image contrast by flattening the distribution to give a more uniform spread of intensities. As a first step we will find the empirical cumulative probability distribution of the number of pixels in each of the intensities bands. This is simple given that we have already determined `N` which is the number of pixels in each intensity band.

```
>> Nc=cumsum(N)./sum(N); %cumulative sum of number of pixels from 0 to 255
```

We now have enough information to be able to transform (via interpolation) from intensities values into cumulative probabilities which will lie within the range 0 and 1. We can then normalize distribution to have extremes of 0 and 1 and multiply by 255 to the values span the full range of possible intensities.

```
>> Jc=interp1([0:255],Nc,GS); %interpolate pixel intensity onto the cdf
>> Jc=Jc-min(Jc(:)); %normalize lowest value to zero
>> Jc=Jc./max(Jc(:)); %normalize largest value to 1
>> Jc=round(Jc*255); %set data in full range [0-255]
```

The values stored in Jc represent our equalized image. We can look at the histogram of equalized intensities and then display the image.

```
>> N = histc(Jc(:),[0:255]); %find number of pixels in each intensity band
>> figure, bar([0:255],N,'histc') %plot histogram
>> xlabel('Intensity [0-255]') %add a x-label
>> ylabel('Number of Pixels') %add a y-label
>> set(gca,'tickdir','out') %make ticks face outwards
```

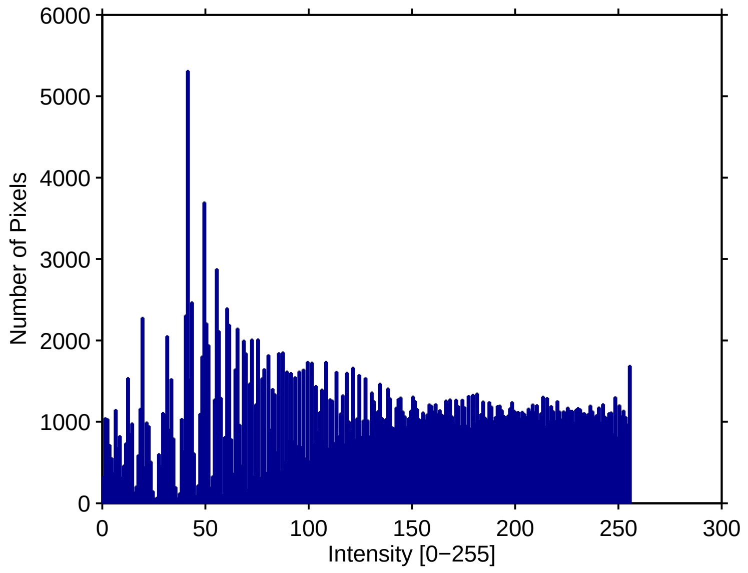


Figure 10.8: *Histogram of equalized image intensities*

```
>> figure, imagesc(Jc) %diplay adjusted image
>> set(gca,'dataaspectratio',[1 1 1]) %set equal x y scaling
>> set(gca,'visible','off') %remove axes
>> colormap gray(256) %change colormap to "gray" with 256 levels
```

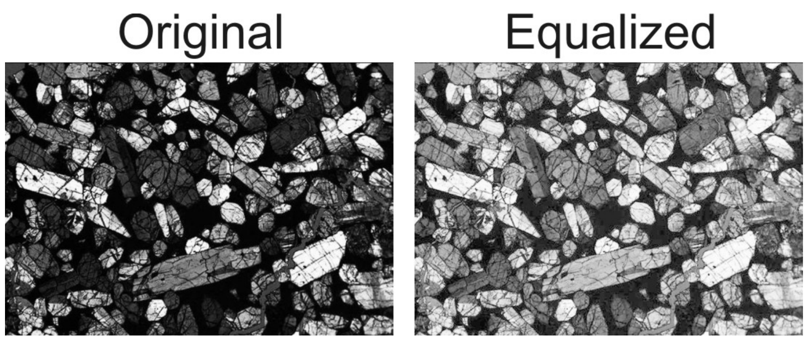


Figure 10.9: *Test image before and after histogram equalization*

10.8 Blurring

It might seem a strange idea but some image processing methods work better if an image is slightly out of focus, a technique known as blurring. We'll return to this idea later, but for the time being we will look at a basic method with which we can blur images called *Gaussian Blurring*. The idea is simple enough, we construct a two-dimensional Gaussian function and convolve it with the image in order that the intensity of pixels are replaced with a weighted average of the their own intensity of the intensity of their neighbors (within a specific spatial extent). This is analogous to forming a weighted moving average of the pixel intensities in both the vertical and horizontal directions. In a similar manner to the moving average, the wider the Gaussian function we construct the more smoothing (blurring) will occur. As with our previous exercises we'll start by loading the image data and convert the RGB pixels into intensities.

```
>> clear all, close all

>> I = imread('thin_section1.jpg'); %load image as 3D RGB array

>> R=squeeze(I(:,:,1)); %obtain copy of red channel information
>> G=squeeze(I(:,:,2)); %obtain copy of green channel information
>> B=squeeze(I(:,:,3)); %obtain copy of blue channel information

>> RGB=[R(:),G(:),B(:)]; %form RGB into a 3 column matrix
>> RGB=double(RGB); %convert to double precision

>> coef=[0.2989; 0.5870; 0.1140] %RGB to intensity transform
>> GS=RGB*coef; %(matrix) multiply RGB and transform coefficiants
>> GS(GS<0)=0; %round up values below 0
>> GS(GS>255)=255; %round down values above 255

>> [r,c]=size(I(:,:,1)) %rows and columns of original image
>> GS=reshape(GS,[r c]); %change from a vector to a matrix

>> imagesc(GS) %diplay intensities with default jet colormap
>> set(gca,'dataaspectratio',[1 1 1]) %set equal x y scaling
>> set(gca,'visible','off') %remove axes
>> colormap gray(256) %change colormap to "gray" with 256 levels
```

Now we'll construct the Gaussian function with which to perform the smoothing. First the width (standard deviation) of the function should be defined, this controls the extent of the blurring because a wider function will average over a larger area of the image for any given pixel. To make things simpler we'll always set the width (assigned as `sigma`) to a positive integer. We'll then evaluate the Gaussian function on a regular grid which spans `-3*sigma` to `3*sigma` in both the x and y directions. Once the function is constructed we can visualize it as a surface.

```
>> sigma=2 %standard deviation of the normal distribution
>> [xf,yf]=meshgrid(-3*sigma:3*sigma,-3*sigma:3*sigma); %xy grid for filter
>> g=exp(-(xf.^2+yf.^2)./(2*sigma.^2)); %normal distribution
>> g=g./sum(g(:)); %normalized so the sum of the filter is 1
>> surf(xf,yf,g) %visualize function as a surface
>> xlabel('x') %label the x-axis
>> ylabel('y') %label the x-axis
>> set(gca,'xlim',[-3*sigma 3*sigma],'ylim',[-3*sigma 3*sigma])
```

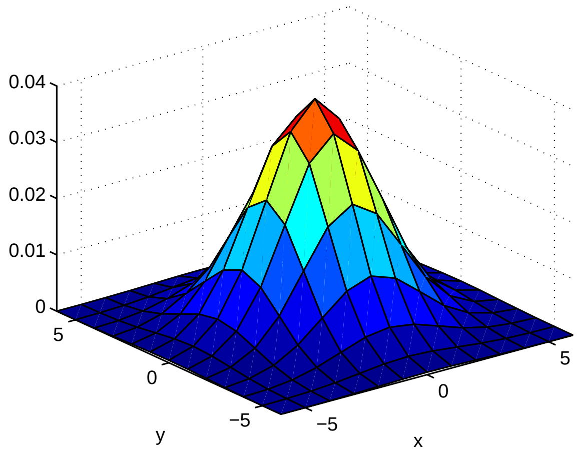


Figure 10.10: *Two-dimensional Gaussian function with a standard deviation of 2.*

To convolve the function and the intensity data is simple using the inbuilt function `conv2` (which performs 2D convolution). We will also use the additional `'valid'` which truncates the final image to remove any edge effects where the function did not sit completely on the image. We can the plot the blurred intensity data in the standard manner.

```
>> GSf=conv2(GS,g,'valid'); %convolve image and filter, remove edge effects
>> figure %create new figure window
>> imagesc(GSf) %diplay image
>> set(gca,'dataaspectratio',[1 1 1]) %set equal x y scaling
>> set(gca,'visible','off') %remove axes
>> colormap gray(256) %change colormap to "gray" with 256 levels
```

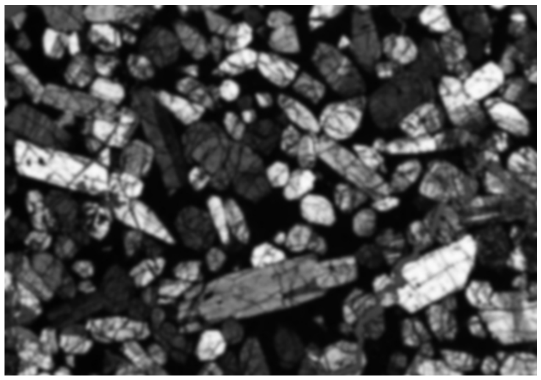


Figure 10.11: *Image after Gaussian blurring by a function with a standard deviation of 2.*

10.9 Edge Detection

As the name suggests the method of edge detection helps us identify possible features in images by isolating regions where the intensity changes rapidly (i.e. where there could be an edge). There are a number of advanced techniques for edge detection (for example canny edge detection) and it is an ongoing field of research. The method we will employ is very simple, detecting sharp changes in intensity by taking the derivative of the image in both the x and y directions. Where we find regions with gradients of a large magnitude it is possible that we have detected an edge. As before we will load out image and transform it into intensity data.

```
>> clear all, close all

>> I = imread('thin_section1.jpg'); %load image as 3D RGB array

>> R=squeeze(I(:,:,1)); %obtain copy of red channel information
>> G=squeeze(I(:,:,2)); %obtain copy of green channel information
>> B=squeeze(I(:,:,3)); %obtain copy of blue channel information

>> RGB=[R(:),G(:),B(:)]; %form RGB into a 3 column matrix
>> RGB=double(RGB); %convert to double precision

>> coef=[0.2989; 0.5870; 0.1140] %RGB to intensity transform
>> GS=RGB*coef; %(matrix) multiply RGB and transform coefficiants
>> GS(GS<0)=0; %round up values below 0
>> GS(GS>255)=255; %round down values above 255

>> [r,c]=size(I(:,:,1)) %rows and columns of original image
>> GS=reshape(GS,[r c]); %change from a vector to a matrix

>> imagesc(GS) %diplay intensities with default jet colormap
>> set(gca,'dataaspectratio',[1 1 1]) %set equal x y scaling
>> set(gca,'visible','off') %remove axes
>> colormap gray(256) %change colormap to "gray" with 256 levels
```

As mentioned above to detect possible edges we need to take the gradient of the intensity image in both the x and y directions. MATLAB has a function `gradient` that will perform the task for us, returning two outputs `Lx` and `Ly`, which are the gradients in the x and y directions, respectively. The magnitude of the gradient is then simply obtained from $\sqrt{Lx^2 + Ly^2}$.

```
>> [Lx,Ly] = gradient(GS) %estimate first-order derivatives in both x and y
>> d=sqrt(Lx.^2+Ly.^2) %gradient magnitudes
```

Of course we have a wide distribution of gradients and we need to choose a threshold to define which values could correspond to real edges. We will perform the this task manual but a number of more advanced techniques can perform it automatically. To make our choice of a cutoff value, we will plot the cumulative probability of the gradient magnitudes.

```
>> cdfplot(d(:))  %plot cumulative distribution of gradient magnitudes
```

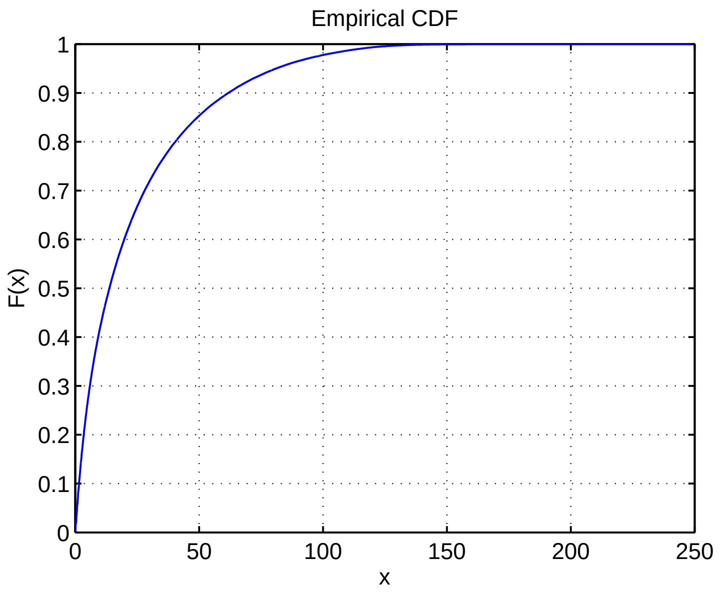


Figure 10.12: *Cumulative distribution of the gradient magnitudes.*

Based on this we will say that only points with a gradient magnitude of 30 or greater will be considered as edges. Therefore we can take our gradient data and set all the values less than 30 to 0 and all those greater than 30 to 255.

```
>> cutoff=30% define cutoff value
>> dc=d; %copy of gradients to which the cutoff will be applied
>> dc(dc<cutoff)=0; %values below cutoff are reduced to zero
>> dc(dc>=cutoff)=255; %values above cutoff are set to maximum
```

We can now visual the edge data as a black and white image.

```
>> figure %create new figure window
>> imagesc(dc) %display image based on gradients
>> set(gca,'dataaspectratio',[1 1 1]) %set equal x y scaling
>> set(gca,'visible','off') %remove axes
>> colormap gray(256) %change colormap to "gray" with 256 levels
```

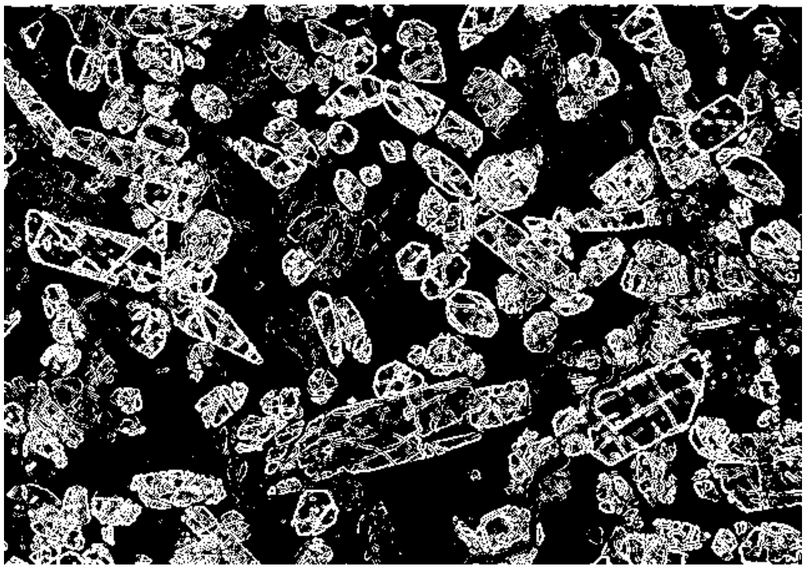


Figure 10.13: *Edges (white) defined by a cutoff gradient magnitude of 30.*

10.10 Exercise: Improved edge detection

In the previous exercise we performed edge detection on a unprocessed image and got a relatively messy result without clearly defined edges. There are a number of preprocessing steps that can help improve the performance of our simple edge detection algorithm. The first is to perform histogram equalization and the second is to blur the image. The idea behind preprocessing with blurring is that small scale features which don't represent true edges will be smoothed out whilst major features (hopefully true edges) will remain. The task of this exercise is to see how preprocessing can improve edge detection in either our example image or an image of your own. You will need to complete the following tasks:

1. Load your image.
2. Convert the image into intensity values.
3. Perform histogram equalization.
4. Perform Gaussian blurring.
5. Perform simple edge detection.

Steps to completing the exercise

This exercise requires a number of steps which follow the previous examples in this chapter. As long as you are careful with variable names you can recycle many of the commands from the earlier examples almost directly. One key idea is to investigate how blurring and modifying the gradient magnitude cutoff can improve the performance of the edge detection algorithm. Therefore you will need to think about:

- The standard deviation of the two-dimensional Gaussian function.
- Trying different cutoff values to find the "optimal" edge detection.

Chapter 11

Final Project

11.1 Sand, Silt and Clay in an Arctic Lake

As a final project we are going to plot and analyze some grain size composition data from a lake in the Arctic. This will require you to undertake all stages of the analysis process from importing your data to plotting your final results. There are a number of steps involved and don't be surprised if this takes some time.

11.1.1 Importing the data

The lake grain size data is stored in the EXCEL file *lakegrainsize.xls*, import the data into MATLAB in a form that you can work with it.

11.1.2 Plotting the data

Plot the grain size data in a ternary plot (look on the MathWorks File-exchange to find functions which will produce ternary plots, or if your feeling confident write your own function). Your final graphic could look something like the following figure.

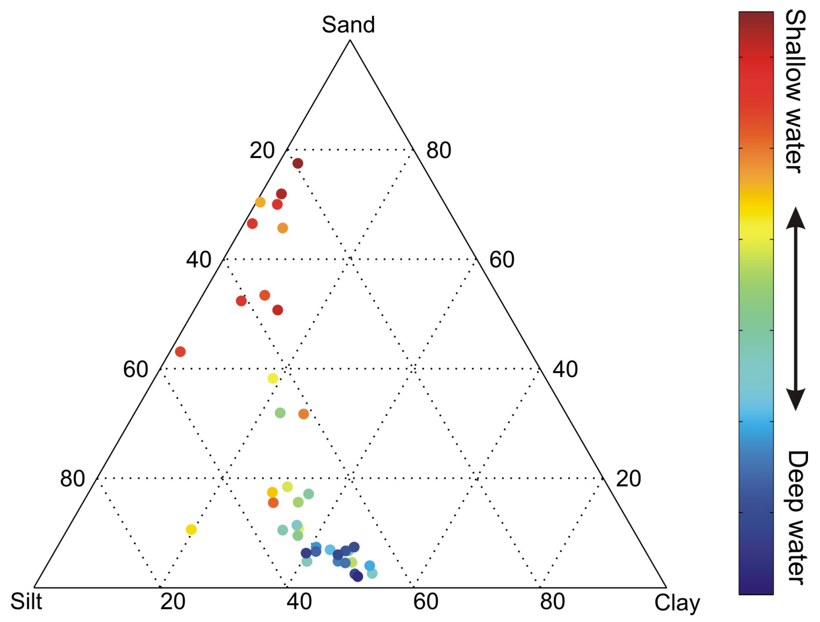


Figure 11.1: *A ternary plot of the Arctic lake grain size data, color-coded according to sample depth.*

11.1.3 Define a regression relationship

Now you must analyze the data to find if there is a relationship between grain size composition and depth. This is not quite as simple as it sounds because we are dealing with closed data, where the composition of each sample must add to 100%. Since we are studying MATLAB rather than statistics, I will guide you through the steps you will need to take to plot the final regression relationship, but it is up to you to write the code.
The statistics we will use follow the ideas of John Aitchison, who developed methods with which to analyze closed data, if you are working with closed data in your own projects it would be a good idea to read his book.

J. Aitchison (2003). The Statistical Analysis of Compositional Data. The Blackburn Press, pp416.

To determine the regression we need to find the linear relationships between:

- The natural log of the ratio $\frac{sand}{clay}$ versus the natural log of lake depth
- The natural log of the ratio $\frac{silt}{clay}$ versus the natural log of lake depth

You can determined the straight-line relationships (i.e. a gradient and an intercept) using the function `polyfit`. For any lake depth you should now be able to predict the log ratios of sand / clay and silt / clay using your regression relationship (look at the function `polyval`).

To convert from log ratios back into sand, silt and clay compositions you can use the equations:

$$sand = \frac{e^{\log(\frac{sand}{clay})}}{e^{\log(\frac{sand}{clay})} + e^{\log(\frac{silt}{clay})} + 1}$$

$$silt = \frac{e^{\log(\frac{silt}{clay})}}{e^{\log(\frac{sand}{clay})} + e^{\log(\frac{silt}{clay})} + 1}$$

$$clay = \frac{1}{e^{\log(\frac{sand}{clay})} + e^{\log(\frac{silt}{clay})} + 1}$$

11.1.4 Plot the regression relationship

Plot the regression line predicting grain size composition for a series of depths through the lake. Your final result should look something like the graphic below.

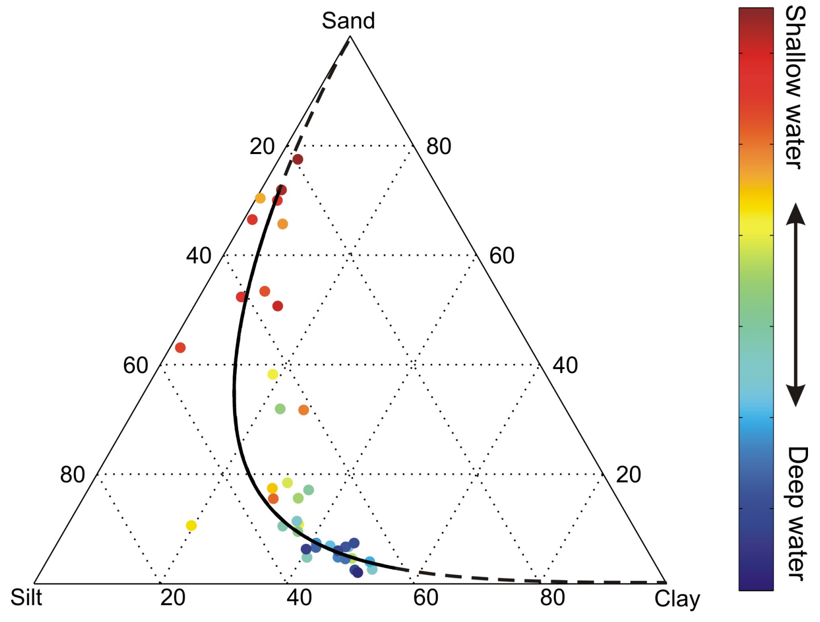


Figure 11.2: *A ternary plot of the Arctic lake grain size data color-coded according to sample depth. The black line shows the regression relationship, the dashes indicate where extrapolation was necessary.*

Steps to completing the exercise

As a review of the material above the basic steps to complete the exercise are as follows:

- Import the data from *Lakegrainsize.xls* and give the variables appropriate names.
- Plot the data in a ternary plot.
- Find the straight-line relationships between the log of depth and the log-ratios of the grain sizes.
- Create an array of depth values.
- Find the log-ratios of the grain sizes for the new log depths according to the straight-line relationships.
- Convert the log-ratio grain sizes back into sand, silt and clay proportions.
- Add the sand, silt and clay proportions from the regression into the ternary plot.

Functions you may find useful

- `xlsread` - Read Microsoft Excel spreadsheet file.
- `ternplot` - Plot ternary phase diagram (available from The Mathworks file-exchange).
- `ternlabel` - label ternary phase diagram (available from The Mathworks file-exchange).
- `log` - Natural logarithm.
- `exp` - Exponential.
- `polyfit` - Polynomial curve fitting.
- `polyval` - Polynomial evaluation.

11.2 Exercise: Bringing it all together with cars and goats

If you still have some time left over you can try this final exercise, it is fun but challenging, some of you might be familiar with the problem because it is quite famous. Again you should think about how to work a solution to the problem before thinking about how specifically to perform the task in MATLAB. This problem is known as the “Monty Hall” problem and was brought to fame when discussed by the world's "most intelligent" person Marilyn vos Savant (she has a quoted IQ of 228). The statement of the problem is as follows:

Suppose you're on a game show, and you're given the choice of three doors: Behind one door is a car; behind the others, goats. You pick a door, say A, and the host, who knows what's behind the doors, opens another door, say C, which has a goat. He then says to you, "Do you want to pick door B?" Is it to your advantage to switch your choice?

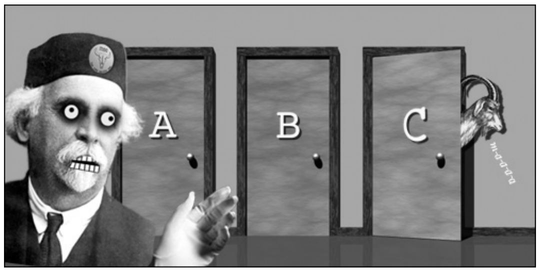


Figure 11.3: *In search of a new car, the player picks a door, say A. The game host then opens one of the other doors, say C, to reveal a goat and offers to let the player pick door B instead of door A.*

.

At first glance it appears the problem comes down to a 50:50 choice and therefore from a probability point of view there is no advantage or disadvantage to switching your choice or sticking with your original choice. However, if you work through the probabilites properly you find that the chances of winning a car are twice as high if you switch compared to sticking with you original choice. This does indeed seem to be a counterintuitive result so we'll attempt to write a MATLAB script which simulates the game. The game can then be run 1 million times (you need a `for` loop to do this) with the various possibilities being played out (you can use relational and logical operators to do this). Does you numerical model get close the result from probability theory? If you are new to programming and MATLAB you may find this exercise very challenging, so don't feel too depressed if you can't complete it.

Steps to completing the exercise

This exercise can be solved with a few inbuilt functions (the list below might give you some clues on how to proceed). To solve the problem take the following steps

- Think.
- Think some more.
- Find the solution.

Functions you may find useful

- `rand` - Uniformly distributed pseudo-random numbers.
- `sort` - Sort in ascending or descending order.

Chapter 12

Exercise Solutions

It is important to realize that the exercises in this course can be solved in a variety of different ways, therefore the solutions I will give are by no means the "definitive answer". I've have attempted to provide solutions which are written in a clear manner in order to help you understand what is happening, this means that the code may be somewhat inefficient and from the point of view of computation speed can certainly be improved.

12.1 Solution: Mmmm, π

There are three main tasks to solve this exercise:

1. Generating the numbers between 0 and 10000 to act as n.
2. Calculating the terms of the series using the values of n.
3. Finally, summing the series and multiplying by 4 to estimate π.

These three tasks can be performed using simple one-line commands. First we need to generate a variable `n` which contains the integers between 0 and 10000:

```
>> n = [0:1:10000];
```

Now we must calculate each term of the series according to expression:

$$\frac{(-1)^n}{2n+1}$$

We'll perform this calculation and place the result in a variable called *terms* (note, my variable names are completely arbitrary, you can use any names you want as long as they follow the rules we discussed earlier).

```
>> terms = (-1).^n./(2*n+1);
```

You'll notice that the order of evaluation is important here and it is necessary to place brackets around the `-1`. Finally, we simply need to add all the terms together using the inbuilt `sum` function and multiply by 4 to make our estimate of π.

```
>> p = sum(terms)*4
```

Putting this all together our script for estimating π using the method of Madhava of Sangamagrama would look like:

```
n = [0:1:10000];
terms = (-1).^n./(2*n+1);
p = sum(terms)*4
```

12.1.1 Golf Solution: Mmmm, π

If you remember, MATLAB Golf involves coding a problem in as few characters as possible. The following lines present one solution in a compact form which takes advantage of a number of MATLAB's shortcuts. It uses 30 characters:

```
>> n=0:1e4
>> sum((-1).^n./(2*n+1))*4
```

12.2 Solution: Mmmm, a second helping of π

To convert our script file into M-function we need to perform three tasks:

1. Add a header line defining the function name (`madhava`), the input (`N`) and output (`p`).
2. Define the values of n to be used as ranging between `0` and `N`
3. Add suitable help and comments

Our original script file looked like:

```
n = [0:1:10000];
terms = (-1).^n./(2*n+1);
p = sum(terms)*4
```

In order to convert this into a M-function we just add an appropriate header line and give `N` as the maximum value of `n`.

```
function p = madhava(N)
n = [0:1:N];
terms = (-1).^n./(2*n+1);
p = sum(terms)*4;
```

Once the function is saved (make sure the filename is the same as the defined function name) you can call it from the MATLAB command window with a defined value of `N`.

```
>> p = madhava(10000)
p =

3.1417
```

We should also add help and comments to explain the purpose of the function and the steps it makes in the calculation.

```
function p = madhava(N)
%madhava - Estimate pi using the Madhava-Leibniz series
%
% Syntax:  p = madhava(N)
%
% Inputs:
% N - the maximum term to be included
%
% Outputs:
% p - estimate of pi
%
% Other m-files required:  none
% Subfunctions:  none
% MAT-files required:  none
%
% Author:  Dave Heslop
% Department of Geosciences, University of Bremen
% email address:  dheslop@uni-bremen.de
% Last revision:  6-Dec-2008

%------------- BEGIN CODE --------------
n = [0:1:N]; %form the integer terms
terms = (-1).^n./(2*n+1); %calculate series terms
p = sum(terms)*4; %sum all terms (to give pi/4) and multiply
%------------- END CODE --------------
```

12.3 Solution: The Geiger counter

This exercise is more complicated than the previous ones. We will solve it in a simple way by testing a number of possible values of N_{true} then selecting the one which gives a value closest to our desired N_{obs}. We'll start by developing the code in a script file with fixed values of

N_{obs} = 8000 cps and τ = 20 x 10^{-6} s. The first thing to do is define these values in suitably named variables:

```
>> Nobs=8000;
>> tau=20e-6;
```

Now we'll create an array of N_{true} values to test, in this case ranging between 1 and 100000:

```
>> Ntest=[1:1e5];
```

To calculate the observed count rate which would be produced by each of the values in `Ntest` we place them in our equation:

```
>> Ncalc=Ntest.*exp(-Ntest.*tau);
```

We now have an array (`Ncalc`) which contains the predicted observed count rate for each of the guessed values in `Ntest`. To find which produces a prediction which is closest to `Nobs` we take the root squared difference between `Ncalc` and `Nobs`:

```
>> Ndiff=sqrt((Ncalc-Nobs).^2);
```

The position within `Ndiff` that has the smallest value tells us the location in `Ntest` which provides the best prediction for `Nobs`. To find this index position we can use the inbuilt function `sort`, this takes an array and reorders it from lowest to highest value and gives a second output with the original index positions of the values before they were sorted.

```
>> [Ndiff_min,idx]=sort(Ndiff);
```

We now know that our best guess for N_{true} lies in `Ntest` at the index given in the first position of `idx`:

```
>> Ntrue=Ntest(idx(1))

Ntrue =

9716
```

So, for an observed count rate of 8000 cps and τ = 20 x 10^{-6} s, we can estimate that N_{true} = 9716 cps. We now have the basic workings of our code and we need to convert it into an M-function. This means we need to think about input and output values. The required inputs will be:

- The observed count rate (called `Nobs`)
- The dead-time (called `tau`)
- The array of values to test (called `Ntest`)

We only need a single output, the predicted value of N_{true}, which will be called `Ntrue`. All this information, including the desired function name, can then be included in a header line followed by the commands we developed above.

```
function Ntrue = geiger(Nobs,tau,Ntest)

Ncalc=Ntest.*exp(-Ntest.*tau);
Ndiff=sqrt((Ncalc-Nobs).^2);
[Ndiff_min,idx]=sort(Ndiff);
Ntrue=Ntest(idx(1));
```

The M-function can be tested with the same problems as we used before, that way we can check that nothing has gone wrong:

```
>> Nobs=8000;
>> tau=20e-6;
>> Ntest=[1:1e5];
>> Ntrue = geiger(Nobs,tau,Ntest)
Ntrue =
9716
```

Finally we should add help lines and comments to explain how the function works.

```
function Ntrue = geiger(Nobs,tau,Ntest)
%geiger - Predict the count rate of a Geiger counter
%
% Syntax:  Ntrue = geiger(Nobs,tau,Ntest)
%
% Inputs:
% Nobs - Observed count rate [cps]
% tau - dead-time [s]
% Ntest - range of count rates to test [cps]
%
% Outputs:
% Ntrue - Predicted true count rate [cps]
%
% Other m-files required:  none
% Subfunctions:  none
% MAT-files required:  none
%
% Author:  Dave Heslop
% Department of Geosciences, University of Bremen
% email address:  dheslop@uni-bremen.de
% Last revision:  6-Dec-2008
```

```
%------------- BEGIN CODE --------------
Ncalc=Ntest.*exp(-Ntest.*tau); %predicted cps
Ndiff=sqrt((Ncalc-Nobs).^2); %rs difference with observed
[Ndiff_min,idx]=sort(Ndiff); %locate smallest difference
Ntrue=Ntest(idx(1)); %select best fitting value
%------------- END CODE --------------
```

12.4 Solution: Classifying sediments

In this exercise you were asked to analyze grain size data from the file *grainsize.mat* in order to find how many samples contained certain amounts of sand, silt and clay. Specifically:

1. How many samples contain more than 50% sand?
2. How many samples contain more than 50% silt?
3. How many samples contain more than 30% silt or more than 25% clay?
4. How many samples contain more than 40% sand, less than 40% silt and more than 40% clay?

The first task is to load the data into MATLAB (plus we will clear the memory before starting:

```
clear all
load grainsize
```

The variable `gs` is now in the memory which contains the data for the sand (first column), silt (second column) and clay (third column) percentages. Each row of the `gs` matrix represents a different sample. We now need to use a combination of relational and logical operators to answer the questions. Lets start with "*How many samples contain more than 50% sand?*". The first thing we need to consider is that the information on sand content is contained in the first column of `gs`, therefore to access the sand content for all the samples we will use the address `gs(:,1)`. We can now find the number of samples with more than 50% sand in two different ways, in the first we will return an array of logical values (`1`s signify samples which meet the criteria and `0`s signify those that don't) and simply add together all the entries:

```
>> output=gs(:,1)>50
>> sum(output)
ans =
19
```

Alternatively, we can use the `find` function to provide an array containing the indices of the samples which met the criteria and then simply find how many entries are in the array using the function `numel`:

```
>> idx=find(gs(:,1)>50)
>> numel(idx)
ans =

19
```

For the remaining questions we will determine an array of logical values and sum the entries together in a single line. Notice how the addressing of the `gs` matrix changes in order to select the sand, silt and clay components as required and the inclusion of logical operators are combinations of criteria have to be tested.

"How many samples contain more than 50% silt?"

```
>> sum(gs(:,2)>50)
ans =
20
```

"How many samples contain more than 30% silt or more than 25% clay?"

```
>> sum(gs(:,2)>30 | gs(:,3)>25)
ans =
90
```

"How many samples contain more than 40% sand, less than 40% silt and more than 40% clay?"

```
>> sum(gs(:,1)>40 & gs(:,3)<40 & gs(:,3)>40)
ans =

0
```

12.5 Solution: The Safe Cracker

The solution to this exercise is a M-function which requires the following basic parts:

1. A name, we'll use `safe_crack`
2. Three inputs, the guessed digits of the combination which we will call `A`, `B` and `C`
3. Testing the combination and sending reports to the screen.
4. An output that we'll call `status` which tells us the state of the safe (`1` corresponds to open, `0` to closed).

Therefore the header line of our M-function will look like:

```
function status=safe_crack(A,B,C)
```

and we can save the function with the filename *safe_crack.m*. We also need to set three digits within the function which will act as the true combination of the safe. We'll call the values Atrue, Btrue and Ctrue and you can give them any single digit values you want. For example:

```
Atrue=6; Btrue=3; Ctrue=7;
```

We also need to record the status of the safe, i.e. open or closed. At this stage we'll assume that the safe is going to open, then if any of the combination digits are wrong we'll set the status of the safe to closed (this might be seem like a strange way to proceed but hopefully you'll see how simple it makes the problem). Now we can set the value of the `status` variable:

```
status=1;
```

Now we must test each of the guessed digits separately using a logical operator and return (an example of this is given in the original exercise). We'll test each value, if a given value is correct we'll send a message to the screen saying so, otherwise we'll send a message that the value is wrong and change the value of `status` to `0`.

```
if A==Atrue %test the first value
   disp('The first value is correct')
else
   disp('The first value is incorrect')
   status=0; %make sure the safe stays closed
end
```

Testing the second and third values follow the same pattern, so our final M-function (including comments and help information) will look like the following code.

```
function status=safe_crack(A,B,C)
%safe_crack - guess the combination of a safe
%
% Syntax:  status=safe_crack(A,B,C)
%
% Inputs:
% A - the first digit of the combination
% B - the second digit of the combination
% C - the third digit of the combination
% Outputs:
% status - state of the safe(1=open, 0=closed=
%
% Other m-files required:  none
% Subfunctions:  none
% MAT-files required:  none
%
% Author:  Dave Heslop
% Department of Geosciences, University of Bremen
% email address:  dheslop@uni-bremen.de
% Last revision:  6-Dec-2008

%------------- BEGIN CODE --------------%
Atrue=6; Btrue=3; Ctrue=7; % set the combination

status=1; %assume the safe will open

if A==Atrue %test the first value
   disp('The first value is correct')
else
   disp('The first value is incorrect')
   status=0; %make sure the safe stays closed
end

if B==Btrue %test the first value
   disp('The second value is correct')
else
   disp('The second value is incorrect')
   status=0; %make sure the safe stays closed
end

if C==Ctrue %test the first value
   disp('The third value is correct')
else
   disp('The third value is incorrect')
   status=0; %make sure the safe stays closed
end
%------------- END CODE --------------%
```

12.6 Solution: An even worse safe

The extension to the previous function to test if the guessed digit is within ±1 of the true digit is simple and we'll save the new function with the name *safe_crack2.m*. Before we tested if the guessed digit was equal to the true digit with the == operator, for the new test we need to check the guess is both greater than or equal to the true digit minus 1 and less than or equal to the true digit plus 1. In the case of the first digit A we can do this with the relational and logical operators:

```
A>=Atrue-1 & A<=Atrue+1
```

This means our final function with modifications for each of the three digits will look like:

```
function status=safe_crack2(A,B,C)
```

```
%safe_crack - guess the combination of a safe within -1 or +1
%
% Syntax:  status=safe_crack(A,B,C)
%
% Inputs:
% A - the first digit of the combination
% B - the second digit of the combination
% C - the third digit of the combination
% Outputs:
% status - state of the safe(1=open, 0=closed=
%
% Other m-files required:  none
% Subfunctions:  none
% MAT-files required:  none
%
% Author:  Dave Heslop
% Department of Geosciences, University of Bremen
% email address:  dheslop@uni-bremen.de
% Last revision:  6-Dec-2008

%------------- BEGIN CODE --------------%
Atrue=6; Btrue=3; Ctrue=7; % set the combination

status=1; %assume the safe will open

if A>=Atrue-1 & A<=Atrue+1 %test the first value
   disp('The first value is correct')
else
   disp('The first value is incorrect')
   status=0; %make sure the safe stays closed
end

if B>=Btrue-1 & B<=Btrue+1 %test the first value
   disp('The second value is correct')
else
   disp('The second value is incorrect')
   status=0; %make sure the safe stays closed
end

if C>=Ctrue-1 & C<=Ctrue+1 %test the first value
   disp('The third value is correct')
else
   disp('The third value is incorrect')
   status=0; %make sure the safe stays closed
end
%------------- END CODE --------------%
```

12.7 Solution: Life's ups and downs in the Logistic Map

This exercise may appear to be more difficult that it actually is. As discussed in the exercise the simplest approach is to write code for a single realization of r and $x_{initial}$ and then extend it to consider multiple cases. For the single case we first need to generate a random value of $x_{initial}$ between 0 and 1 and r between 0 and 4. To do this we can use the `rand` function which gives uniformly distributed random numbers in the range 0–1:

```
x=rand(1);
r=rand(1)*4;
```

Once we have these starting values we can employ a loop to iterate the logistic equation a total of 2000 times, each time updating the existing value of `x`.

```
for i=1:2000
   x=r.*x*(1-x);
end
```

The value of `x` will now represent x_{final} for our specific case of r and $x_{initial}$. To extend this code to a large number of realizations so that we can plot the logistic map we need to generate a large number of values of r and $x_{initial}$ (in this example we'll use 10000). Then we loop through each realization, iterating the logistic equation 2000 times to produce all the values of $x_{initial}$. You'll see that to iterate the equation for all the different cases we need to use two loops inside each other:

```
N=1e4 %total number of realizations
x=rand(N,1); %generate the initial populations
r=rand(N,1)*4; %generate the values of r

for j=1:N %loop through all the realizations
   for i=1:2000 %iterate the logistic equation
      x(j)=r(j).*x(j)*(1-x(j)); %update the population
   end
end
```

Finally we just need to plot the data and label the axes:

```
plot(r,x,'.')
ylabel('x_{final}')
xlabel('r')
```

12.8 Solution: Plotting elevation data

The exercise requires loading and plotting the elevation data from the file *Hawaii.mat*. The first task is to load the data and look at the variables:

```
>> load Hawaii
```

The first thing we need to do is make regularly spaced arrays of latitude and longitude, which pass from their minimum to maximum values in steps of 0.1^o. To do this we can use the functions `min` and `max`, we'll call the arrays `X` (for the longitude) and `Y` (for the latitude).

```
>> X=[min(longitude):0.1:max(longitude)];
>> Y=[min(latitude):0.1:max(latitude)];
```

Now we'll make regular grids called `Xgrid` and `Ygrid` from the `X` and `Y` arrays using the `meshgrid` function and then interpolate the `elevation` data onto the grids.

```
>> [Xgrid,Ygrid]=meshgrid(X,Y);
>> Zgrid=griddata(longitude,latitude,elevation,Xgrid,Ygrid);
```

To plot the topographic surface we need to make a copy of `Zgrid`, which we'll call `topo`, and the replace all the values below zero with `0`. This basically means we are taking all the points below sealevel and replacing them with zeros. Then we'll plot the data using `surf` and add labels to the axes.

```
>> topo=Zgrid; %make a copy of the data
>> idx=find(topo<0); %find the indicies of the points below zero
>> topo(idx)=0; %replace these points with zeros
>> figure %create a new figure window
>> surf(Xgrid,Ygrid,topo) %plot the data grids as a surface
>> xlabel('Longitude [o]') %add a x-axis label
>> ylabel('Latitude [o]') %add a y-axis label
>> zlabel('Height [m]') %add a z-axis label
```

Producing a bathymetric map follows exactly the same ideas, except this time when we make a copy of the data we will replace the height values greater than zero with `0`.

```
>> bath=Zgrid; %make a copy of the data
>> idx=find(bath>0); %find the indicies of the points above zero
>> bath(idx)=0; %replace these points with zeros
>> figure %create a new figure window
>> surf(Xgrid,Ygrid,bath) %plot the data grids as a surface
>> xlabel('Longitude [o]') %add a x-axis label
>> ylabel('Latitude [o]') %add a y-axis label
>> zlabel('Depth [m]') %add a z-axis label
```

12.9 Solution: The Geiger counter

We can obtain our best estimate by finding the value of N_{true} which produced the minimum squared difference of the count rate equation with the observed count rate. We can write a inline function to measure this difference, it will use given values of N_{true} and τ to calculate a guess of N_{obs} using the count rate equation and then take the squared difference with the actual value of N_{obs}.

```
>> f=inline('(Ntrue.*exp(-Ntrue.*tau)-Nobs)^2','Ntrue','Nobs','tau')
```

The inline function `f` can now be called in `fminsearch` where we also need to provide values for N_{obs}, τ and an initial guess of N_{true} (in this case we'll use a value of 9000):

```
>> x = fminsearch(@(Ntrue) f(Ntrue,20e-6,8000),9000)
```

12.10 Solution: Sand, Silt and Clay in an Arctic Lake

This exercise brings together all the aspects of data processing with MATLAB, you are required to import raw data, plot it, perform an analysis and plot the final results. The stages of the exercise have been described in detail in the main text, so here we will just concentrate on the required MATLAB steps. The first thing to do is import the data from the EXCEL workbook *Lakegrainsize.xls*. When we looked at the workbook in excel we see there are four columns of data for depth, sand, silt and clay, which are held in the cells A2:A40, B2:B40, C2:C40 and D2:D40, respectively. With this information we can enter commands into a script file which will import the data into MATLAB (you will need to change the path for the EXCEL book according to where it is stored on your computer):

```
>> clear all, close all
>> depth=xlsread('G:\GLOMAR\final_project\Lakegrainsize.xls','A2:A40');
>> sand=xlsread('G:\GLOMAR\final_project\Lakegrainsize.xls','B2:B40');
>> silt=xlsread('G:\GLOMAR\final_project\Lakegrainsize.xls','C2:C40');
>> clay=xlsread('G:\GLOMAR\final_project\Lakegrainsize.xls','D2:D40');
```

We now have the four variables `depth`, `sand`, `silt` and `clay` in the memory and our next task is to plot the grain size data in a ternary plot. To create the ternary plot I have chosen to use the functions `ternplot` and `ternlabel` which can be downloaded from the The Mathworks File-Exchange, however, there are alternatives are available which you could also use.

```
>> ternplot(clay,sand,silt,'.b ') %Ternary plot with blue data points
>> hold on
>> ternlabel('Clay','Silt','Sand') %Add labels to the ternary plot
```

The next steps are probably the most difficult as we need to form regression relationships between the lake depth and the grain size composition. As discussed in the exercise text the first thing to do is form the log-ratios of the grain size components using `clay` as the denominator variable:

```
>> sand_clay=log(sand./clay)
>> silt_clay=log(silt./clay)
```

Now we must find the straight-line relationship between the log of depth and the log-ratios of the grain size components. There are many ways to do this but one of the simplest is to use the function `polyfit`, which requires 3 inputs, the independent data which in this case is the log of `depth`, the dependant data which in this case is the one of the log-ratios, and finally the order of the polynomial to be fitted (we want a straight-line which is order 1). The output variables from `polyfit` store the structure of the polynomial fit, which for a straight-line is the values of the gradient and the intercept.

```
>> p_sand_clay=polyfit(log(depth),sand_clay,1);
>> p_silt_clay=polyfit(log(depth),silt_clay,1);
```

In order to draw our regression line representing the relationship between lake depth and grain size composition we must first define a series of depths that we'll call `D`. Then we'll find the grain size compositions for the log of these depths according our derived regression relationships. This can be done using the function `polyval` which will evaluate a polynomial with known coefficients for a given value. You can see in the commands below we supply `polyval` with the polynomial coefficients calculated in the previous step and ask it to determine values at the logs of the new depth points.

```
>> D=[1:1:500]; %series of new depths for the model
>> sand_clay_hat=polyval(p_sand_clay,log(D)); %log(sand/clay) model
>> silt_clay_hat=polyval(p_silt_clay,log(D)); %log(silt/clay) model
```

Our predictions for the grain sizes are still in the form of log-ratios, but we can convert them back into sand, silt and clay proportions using the equations given in the exercise:

```
>> sand_hat=exp(sand_clay_hat)./(exp(sand_clay_hat)+exp(silt_clay_hat)+1);
>> silt_hat=exp(silt_clay_hat)./(exp(sand_clay_hat)+exp(silt_clay_hat)+1);
>> clay_hat=1./(exp(sand_clay_hat)+exp(silt_clay_hat)+1);
```

Finally, we just need to add our new regression points as a line in the ternary plot:

```
>> ternplot(clay_hat,sand_hat,silt_hat,'r') %Add the regression to the plot
```

12.11 Solution: Bringing it all together with cars and goats

The best way to proceed with the problem is to think how we would play the game once and build that into a repeating process so that we can play millions of times and examine the probability of the different results. Lets start with the basic mechanics of the game which we can break down into parts, firstly we need to decide on the combination of goats and car which will be behind the doors. To make this a bit easier we will always use the sequence *car-goat-goat* and randomly generate the door numbers they are behind. This means we need to produce the numbers 1,2 and 3 in an random order, which can be done by generating three random numbers and finding the indices which will sort them into ascending (or descending) order, for example:

```
>> r=rand(3,1);
>> [r,door]=sort(r);
>> door
door =

3
1
2
```

This means that for this specific game the car would be behind door 3 and goats would be behind doors 1 and 2. The same process can be combined into a single line:

```
>>[door,door]=sort(rand(3,1));
```

We can perform a similar process to obtain the first guess of the contestant, again we'll generate the numbers 1,2 and 3 in a random order and take the first number as the guess:

```
>> [guess,guess]=sort(rand(3,1));
>> guess=guess(1)
guess =

1
```

So in this case the contestants first guess would be that the car is behind door 1. Now we can test the first case, did the contestant correctly guess the door in the first instance, if they did we then have to decide what happens at the next stage, will they stick with their original choice, in which case they will win the car, or will they switch, in which case they will win a goat. To represent the choice of wether to switch doors or not we'll generate a random number between 0 and 1, if the value is <0.5 the contestant sticks with the original door, of the number is >0.5 they will switch.

```
switch_door=rand(1);

if guess==doors(1) %the correct door was chosen at the beginning
```

```
   if switch_door<0.5 %if the number is <0.5 the door isn't switched
      disp('You win a car')
   end
end
```

The opposite situation is a simple extension of this, if the contestant did not originally choose the correct door then they will only win a car if the value of `switch_door` is greater then 0.5. Therefore both possibilities to win the car can be written as:

```
switch_door=rand(1);

if guess==doors(1) %the correct door was chosen at the beginning
   if switch_door<0.5 %if the number is <0.5 the door isn't switched
      disp('You win a car')
   end
end

if guess~=doors(1) %an incorrect door was chosen at the beginning
   if switch_door>0.5 %if the number is >0.5 the door is switched
      disp('You win a car')
   end
end
```

We now have the structure of the code with which we can play a single game, but we need to make adjustments so that we can play it repeatedly. If you remember our original question we wanted to know if we are more likely to win if we switch doors (which we'll call a *type1* win) rather than sticking with the same door (a *type2* win). All we need to do is place the game simulation within loop so that we can run it a number of times and add up the number of *type1* and *type2* wins. Finally, we can find the relative probabilities by calculating the total number of *type1* wins compared to the total number of wins.

```
type1_total=0;
type2_total=0;

for iter=1:1e5

   [doors,doors]=sort(rand(3,1));
   [guess,guess]=sort(rand(3,1));
   guess=guess(1);

   switch_door=rand(1);

   if guess==doors(1) %the correct door was chosen at the beginning
      if switch_door<0.5 %if the number is <0.5 the door isn't switched
         type2_total=type2_total+1;
      end
   end
   if guess~=doors(1) %an incorrect door was chosen at the beginning
      if switch_door<0.5 %if the number is >0.5 the door is switched
         type1_total=type1_total+1;
      end
   end

end

prob=type1_total./(type1_total+type2_total)
```