ESE 3.09: Numerical Methods 1

Dr David Ham

Spring Term 2012

Introduction

0.1 Contact details

Lecturer:	Dr David Ham
Email:	David.Ham@imperial.ac.uk
Office:	3.55
Phone:	020 7594 6439 (x46439)

0.2 Aims

The course will cover the basic techniques which underlie the computational mathematical techniques in common use in the Earth sciences. It will provide an overview of the capabilities and limitations of scientific computation including the concepts of error and conditioning. The course will cover techniques applicable to both observational data and simulation and will have an emphasis on the practical implementation of the methods studied in the Python programming language.

0.3 Objectives

By the end of this course, you should be able to:

- write Python code for a number of important and useful mathematical algorithms
- understand the sources of error in algorithms
- be able to apply numerical techniques to solve problems which occur in geoscience such as curve fitting, solving systems of equations and root finding

0.4 Assumed knowledge

The prerequisites for this module are Maths Methods 1 and 2, and Programming for Geoscientists. This module will use and build upon the material from those courses extensively. Particular use will be made of:

Maths methods 1 Newton-Raphson iteration, matrices and vectors.

Maths methods 2 Taylor series.

Programming for Geoscientists writing and debugging short programmes in Python.

0.5 Lecture plan

The module will cover the numerical implementation of many of the areas of mathematics which will have been covered at school and in previous university modules.

We will start by looking at basic arithmetic: addition, subtraction and multiplication. We will examine the impact of using binary arithmetic on a computer with finite number lengths.

The following three weeks will focus on the numerical implementation of calculus: first differentiation and then integration. Week 5, root finding, is concerned with numerical methods for automatically solving equations on a computer. In weeks 6 and 7 we will examine numerical linear algebra: calculation and equation solving with matrices and vectors. This leads up to the final week in which we will learn to solve overdetermined systems in a least squares sense and apply this knowledge to the problem of fitting curves to scientific data.

Lecture 1 Binary arithmetic

Lecture 2 Floating point numbers, round-off error and Approximating functions

Lecture 3 Taylor series and numerical integration

Lecture 4 Numerical differentiation

Lecture 5 Root finding

Lecture 6 Numerical linear algebra

Lecture 7 LU factorisation

Lecture 8 Curve fitting and least squares solutions

0.6 Course text

The course will be primarily taught from lecture notes, which will be issued each week. You will also find it useful to refer back to the course text for Programming for Geoscientists:

Langtangen, Hans Petter *A primer on scientific computing with Python*, Springer, 2009.

0.7 Assessment

0.7.1 Exam

There will be a question associated with this module on one of the combined exams. As usual, a choice of two questions will be provided of which students taking this module must answer at least one. The exam will focus on the *mathematical* parts of the module.

0.7.2 Programming exercise

The one piece of assessed coursework will be a programming exercise which will take place at some point during the reading week at the end of the term. You will be given some mathematical expressions or algorithms and asked to write programmes in Python which implement them. The programming exercise will focus on testing the *programming* parts of the module.

0.8 Coursework

The lecture notes contain exercises and the last hour of each lecture will be devoted to working on exercises. The lecturer and demonstrators will be available to provide help and immediate feedback. In addition, every two weeks you should hand in the solutions to those two weeks exercises in order to receive written feedback. The submission dates for the exercises have already been set up on ESESIS.

It is likely that you will need to devote additional time to completing the exercises outside of the lectures.

The coursework is not assessable. The solutions you hand in will be *corrected* but no numerical marks will be given. The coursework is therefore your opportunity to gain *feedback* on your progress with no penalty for failure. You are encouraged to hand in your best attempt at a solution even if it is wrong in some way so that you receive feedback in the form of corrections.

0.8.1 The course directory

There are various example programs, code and data to assist with the exercises. These are to be found in the `/numerical-methods-1` directory on `student.ese.ic.ac.uk`, which is the machine you log in to from the department labs. The exercises will indicate which files you need. You should copy the files to the directory you are working in before using them. It is a good idea to create a new working directory for each topic covered in the course so as to maintain some order in your work and reduce possibilities for confusion.

0.9 What to hand in

There are various sorts of question in this module, each of which has different outputs which need to be handed in. ESESIS has been configured to expect *both* a written part and an online part to each set of exercises. Please follow the instructions in the subsequent paragraphs carefully in determining what to hand in online and what to hand in on paper.

0.9.1 Programs

Some questions will ask you to write a program, or a module containing particular functions. For these questions, you should submit the source code *online* but also submit *printouts*. The demonstrator will run your source code on a computer to check for any errors and will provide written feedback on the printouts.

The easiest way to obtain printouts of your programs is using `a2ps`. For example, if you have a Python file named `foo.py` then:

```
> a2ps foo.py
```

will print a nicely formatted version of `foo.py` to the college print system.

ESESIS will only allow you to upload a single file for each assignment so you need to bundle together the files you intend to upload in a single archive file. The easiest way to do this is to place all the files you intend to upload into a single directory. For example, suppose you had placed all the files you wish to upload for weeks 1 and 2 in a directory called `week_1_2`. From the *directory above* `week_1_2` you could then type:

```
> tar cfvz week_1_2.tgz week_1_2
```

This will create `week_1_2.tgz` containing the whole `week_1_2` directory. You can list the contents of the `tgz` file to double check what you are submitting using:

```
> tar tfvz week_1_2.tgz
```

See section 0.10 below for guidance as to what should be in the programs you hand in.

0.9.2 Computer output

Some questions require you to produce plots or graphs using Python. You should print these out and submit them as hard copy. Always ensure that your axes are labelled and that there is a legend showing what the lines mean. The `xlabel`, `ylabel` and `legend` Python commands are particularly useful for this. You should also *neatly* write a short caption indicating what the plot shows and which exercise it answers. Do not leave the poor demonstrator to guess why you are handing in this plot!

If you wish to embed your plots in a typed document in LaTeX or a word processor, you may do so but this is not required and it is perfectly acceptable to neatly write on the printouts.

0.9.3 Mathematical calculations and proofs

Where a question asks you to make a mathematical calculation or a proof, you should write out your answer showing your working. Students often have particular difficulty working out how much detail to write down in proof questions. You should take the proof of theorem 3.3 as an example of a good proof answer.

0.10 Components of a correct program

0.10.1 Interfaces

Software is comprised of components which have to work together. This is achieved by strictly sticking to specified interfaces. Coursework questions will specify module and function names and interfaces. They may also specify particular exceptions which are to be raised in the event of erroneous input or the failure of an algorithm.

0.10.2 Comments and style

A program is not just a series of instructions for a computer, it's also a medium for communicating your ideas to the next programmer. You are expected to provide a docstring for every function describing what that function does. In addition, your code should be commented to a standard that it would be immediately obvious to another student what each line of the program does. Be careful, though, over-commenting can obscure the meaning of a program just as much as too few comments.

A consistent and clear programming style is also essential to readability. Ask yourself do the variable names make sense? Is the indentation consistent? Does the way in which the program is split into modules and functions aid or obscure understanding?

0.10.3 Functionality

Ensure that you carefully read the question and that your program does exactly what is required of it. Use test data to ensure the right answer and, where available, test your program against the "official" version in Numpy or Scitools.

0.11 Plagiarism

You are encouraged to read widely online and to help each other in solving coding problems, but directly copying code from other sources without attribution is plagiarism just like copying text. It's also surprisingly easy to tell when code has been copied: an experienced programmer will see differences of coding style just as easily as differences of writing style in text.

We may also ask you to explain what a particular line of your code does!

Remember, the coursework is not assessable so you gain precisely no marks by cheating. The in-class programming exercise, however, *is* assessed so *you* need to be able to write programs by yourself to pass this module.

0.12 Misuse of computers during lectures

It is an unfair distraction to the students around you if you are web surfing, checking email or browsing social media during the lectures. Students caught engaging in this behaviour will be asked to leave the lecture.

0.13 Notation

These notes contain many Python programmes and snippets of Python code. In common with the course text "A Primer on Scientific Programming with Python" by Hans Petter Langtangen, Python code is placed on a pale blue background with complete programmes distinguished by a coloured bar at the left. For example, the famous one-line program with which all texts start is rendered:

```
#!/usr/bin/env python
print 'Hello World!'
```

To avoid endless repetition, the initial line `#!/usr/bin/env python` will be omitted in the examples through the text. Occasionally, examples of input on the Linux command line will be given. These are distinguished by a grey rather than a blue background. For example:

```
> a2ps foo.py
```

0.14 Extension material*

Some parts of these notes extend the basic course material. Extension material will be marked with an asterisk (*). These sections are present for the benefit of those students who are interested in taking the module a little further, but don't worry, they won't be on the exam! There are a few exercises marked with a double asterisk (**). This is particularly challenging material to be attempted by the very brave.

Chapter 1

Binary arithmetic

If we are to use computers to do our arithmetic, one of the most important details of which we must be aware is how computers store numbers and do arithmetic on them. As in pen-and-paper mathematics, there are different sorts of number system. For example, in the mathematical world, we are familiar with the integers (whole numbers), rational numbers (fractions) and real numbers (the whole of the number line), just to name a few. We know that the properties of arithmetic operations are different too, for example $\sqrt{2}$ cannot be evaluated in the integers or the rationals, only in the reals. Conversely, if we have the list ["cat", "dog", "goldfish", "giraffe"], it makes no sense to ask "what is in the 1.5th place in this list". Another way of saying "it makes no sense to ask" is to say that this operation is not defined. This is an important concept in computer arithmetic, to which we will return later. However for this lecture, we will concentrate on integers, *the whole numbers*.

1.1 Binary integer arithmetic

In the binary system, instead of digits we have binary digits, or *bits*. Whereas a decimal digit can take any one of the ten values in the set $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, a binary digit can only take the values 0 or 1. All computers in current use operate on the binary system for one simple reason: an electrical circuit exists in two basic states, off and on. Binary arithmetic is the natural result of representing numbers as sets of electrical circuits switching off and on.

1.1.1 Counting

So if 0 and 1 are the only digits available, how do we write the number two in base two? The analogous question in the decimal system is "how do you write down the number ten". The answer, which you have known since primary school, is to use successive powers of the base in a place value system. So ten is written as 10 in base ten, where this notation is understood to mean "one times ten plus zero times one". So the number two is written in base two as 10_2, which means "one times *two* plus

zero times one". Notice the subscript 2 which is used to distinguish binary numbers from decimal ones.

We can also write 11_2 and understand that this means "one times two plus one times one", in other words, 3. What about four? This is analogous to the number one hundred in base ten which is written 100. Four is written as 100_2 which is understood as "one times two squared plus zero times two plus zero times one". Following this pattern, we can write the numbers from zero to, say, nineteen in base 2:

$$\begin{aligned}
0 &= 0_2 & 10 &= 1010_2 \\
1 &= 1_2 & 11 &= 1011_2 \\
2 &= 10_2 & 12 &= 1100_2 \\
3 &= 11_2 & 13 &= 1101_2 \\
4 &= 100_2 & 14 &= 1110_2 \\
5 &= 101_2 & 15 &= 1111_2 \\
6 &= 110_2 & 16 &= 10000_2 \\
7 &= 111_2 & 17 &= 10001_2 \\
8 &= 1000_2 & 18 &= 10010_2 \\
9 &= 1001_2 & 19 &= 10011_2
\end{aligned}$$

1.1.2 Addition

The addition rules for binary numbers are directly analogous to those for decimal numbers: when any column overflows then an extra value is carried to the next column. For example, if we calculate two plus three, we have:

$$\begin{array}{r} 10_2 \\ 11_2+ \\ \hline 101_2 \end{array}$$

Notice that adding the two ones in the twos column produces zero twos plus a carried 1 in the fours column.

Exercise 1.1. Compute the following by first converting to binary:

1. 7+3
2. 15+1

1.1.3 Multiplication

Multiplication is achieved in exactly the same manner as long multiplication for decimal numbers. Of course in binary this is a particularly easy operation since at each

step, the first operand is multiplied by either 1 or 0. For example, five times five is written:

$$\begin{array}{r} 101_2 \\ 101_2 \times \\ \hline 101_2 \\ 000_2 \\ 101_2 \quad + \\ \hline 11001_2 \end{array}$$

Notice the carry in the eights place when calculating the final summation.

Exercise 1.2. Calculate the following by first converting to binary:

1. 5×3
2. 6×6

Exercise 1.3. Calculate 3×2 and 3×4 in binary. What is the rule for multiplying by a power of 2 in binary?

1.2 Some notation

There is a standard set of notation associated with the different mathematical number systems which it is useful to be aware of:

- $\mathbb{Z}$ integers: the whole numbers.
- $\mathbb{Q}$ rationals: $\frac{a}{b}$ for $a, b \in \mathbb{Z}$.
- $\mathbb{R}$ reals: the whole number line.
- $\mathbb{C}$ complex numbers: $a + bi$ for $a, b \in \mathbb{R}$ where $i = \sqrt{-1}$.

1.3 Unsigned integers

A feature of all of the sets of numbers in the preceding section is that they are infinite: there are an infinite number of possible integers, and there are an infinite number of rationals and reals even between 0 and 1. However, computers have a fixed amount of storage and are built to work on binary numbers with a fixed size. Let's first consider integers. A computer integer is expressed as a fixed number of binary digits, or *bits*. Let's confine ourselves to the positive numbers and zero, a set referred to in computing as the unsigned integers. An unsigned 3 bit integer represents numbers

as follows:

2^2	2^1	2^0	
4	2	1	decimal
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

So a 3 bit unsigned integer is only capable of representing the numbers from 0 to 7. Equivalently, an n bit unsigned integer can contain the numbers from 0 to $2^n - 1$.

1.4 Signed integers

Of course we will want to represent negative numbers too. The most straightforward way to do this, and the solution which was common in the early days of computing, is to simply add one bit to the start of the number and designate 0 for + and 1 for -. In this system, our 3 bit unsigned integer becomes a 4 bit signed integer:

	2^2	2^1	2^0	
sign	4	2	1	signed decimal
1	1	1	1	-7
1	1	1	0	-6
1	1	0	1	-5
1	1	0	0	-4
1	0	1	1	-3
1	0	1	0	-2
1	0	0	1	-1
1	0	0	0	-0
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7

We can immediately observe two peculiarities with this system. First, there are two zeros with different signs. Second, the binary rule for counting is different for positive and negative numbers. For positive numbers, the rule is the rule we would expect. However if k is negative then $k + 1$ has to be evaluated as $-(-k - 1)$. This way of storing integers is known as *sign magnitude*.

The approach which is taken for integers on almost all modern computer systems is known as *two's compliment*. In this case, the number -1 is defined as the bit pattern

such that adding 1 produces 0. This has the result that -1 is represented by a bit pattern of all 1s. We can illustrate this for a 4 bit signed integer, as follows:

$$\begin{array}{r} 1111_2 \\ 1_2+ \\ \hline \not{1}0000_2 \end{array}$$

Note that the carried 1 in the 2^5 column is dropped because our number format only supports 4 columns. The 4 bit two's compliment signed integers are:

	2^2	2^1	2^0	
sign	4	2	1	signed decimal
1	0	0	0	−8
1	0	0	1	−7
1	0	1	0	−6
1	0	1	1	−5
1	1	0	0	−4
1	1	0	1	−3
1	1	1	0	−2
1	1	1	1	−1
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7

We can see in this representation that there is a single zero and that the rule for counting (and therefore for addition) is the same for positive and negative numbers. It turns out that the rule for multiplication is also exactly the same for signed and unsigned arithmetic, but we won't cover the details here. In fact, the machine implementation of unsigned and two's compliment signed n bit integer addition and subtraction is exactly the same: it's only the *interpretation* of some of the numbers as negative which differs.

1.4.1 Calculating the negative of a number

Two's complement arithmetic also gives us an easy mechanism for finding the bit pattern corresponding to a negative number. The hint is in the word "complement". The complement of a binary number is the number which results from changing all the zeros to ones and all the ones to zeros. So in 4 bit binary, the complement of 2 (0010_2) is 1101_2. If we add these two numbers then we have 1111_2, which is -1. If we think about this for just a second, it becomes obvious that the sum of a number and its binary complement is always -1. If we write $\bar{x}$ for the complement of x, then:

$$x + \bar{x} = -1$$

with a little rearrangement, this becomes:

$$-x = \bar{x} + 1$$

So we now have an algorithm for finding the negative of x:

1. Compute $\bar{x}$ by changing all the zeros in x to one and all the ones to zero.
2. Add one to $\bar{x}$.

To gain some intuition into this result, it is useful to note that $-x$ is the number which, when added to x, yields 0. In contrast, $\bar{x}$ when added to x gives -1. Consequently, $-x$ must be one more than x.

Exercise 1.4. Calculate $5 - 6$ by rewriting the problem as $5 + (-6)$, and rewriting both operands as signed 4 bit binary integers. Next, add using the rules for unsigned addition from section 1.1.2. Verify that your answer, when interpreted as a signed integer, is -1

Exercise 1.5. Calculate 2×-2 by first converting both operands to signed integers, then multiply using the rules for unsigned integers from section 1.1.3. Verify that your answer, when interpreted as a signed integer, is -4.

1.4.2 Determining integer size

How do we know what is the biggest integer available on our computer? In Python there is a special variable for this in the `sys` package. The largest ordinary integer in python can be found with:

```
import sys
print sys.maxint
```

So how big is that in bytes? We can assume that `sys.maxint` must be $2^{n-1} - 1$ where n is the number of bits in the number. So:

$$b = 2^{n-1} - 1$$
$$\log_2(b) = \log_2\left(2^{n-1} - 1\right)$$

Now for large n, $\log_2\left(2^{n-1} - 1\right) \approx \log_2\left(2^{n-1}\right)$ so:

$$\log_2(b) \approx \log_2\left(2^{n-1}\right)$$
$$= n - 1$$

Happily, the numeric Python module `numpy` contains a function for taking the base 2 logarithm of a number so we can try:

```
import sys
import numpy
print numpy.log2(sys.maxint)
```

On my computer, this produces `63.0` so my integers are 64 bit, a sensible answer. The other answer which you may well see on a modern computer is 32 bit. 32 and 64 are the only default integer bit lengths in current use.

1.4.3 Integer overflow

So what is `sys.maxint+1`? The answer depends on the computer system. In Python, data types are very flexible so the answer is a larger integer type:

```
import sys
sys.maxint+1
Out[1]: 9223372036854775808L
```

Note the `L` at the end of the answer. This indicates a long integer, which is a Python data type with essentially no maximum size. Long integers are very flexible but, because they are not directly handled by the hardware, computations with them can be very slow.

With a little trickery, we can force Python to calculate `sys.maxint+1` in hardware. To do this, we use an `array`. We will encounter arrays again later but for now, the only thing we need to know is that the data type of an array is fixed when it is created so Python won't perform the promotion to long integer:

```
import sys
import numpy
numpy.array(sys.maxint)+1
Out[2]: -9223372036854775808
```

This is the negative of the answer we expect! What has happened? Remember that for two's compliment signed integers, the rules for addition are the same as for unsigned integers. Let's look back to our example of 4-bit integers. The largest signed value is 7, or 0111_2. Let's add 1:

$$\begin{array}{r} 0111_2 \\ 1_2+ \\ \hline 1000_2 \end{array}$$

As an *unsigned* integer, 1000_2 is 8. However, in two's compliment it's -8. In other words, integer addition "wraps around". This process is known as overflow and accounting for it is important in any algorithm which uses large integers.

Exercise 1.6. What is `numpy.array(sys.maxint)*2`? Why is this the case?

Chapter 2

Floating point numbers

2.1 Floating point numbers

Fixed sized numbers present even stronger problems for representing real numbers than for integers, as there are infinitely many reals between every two integers. It is also fairly common to wish to represent real numbers at a huge range of scales. For example, the angular velocity of the earth is approximately 7.3×10^{-5} rad/s while the age of the Earth is around 1.4×10^{17} s.

In fact, scientific notation gives us a hint as to how computers deal with these numbers. A *floating point number* is represented as follows:

$$s1.m \times 2^{e-b} \tag{2.1}$$

Where: s is the sign (0 for $+$, $-$ for minus), m is known as the *mantissa* and e is the *exponent*. b is the exponent *bias* which is present to enable the exponent to take negative values, since e is represented as an unsigned integer.

Lets unpack this notation a little. Why do we know that the first digit of the mantissa is always a 1? Well actually scientific notation is ambiguous: 3.0×10^0 is the same as 0.3×10^1. However it's much more efficient to have a single storage form for each number. Floating point arithmetic therefore adopts the convention that the exponent will be set so that the digit before the point is non-zero. This is known as the *normal form* of the number. For example, the normal form of 3.0 is 3.0×10^0. Of course computer floating point arithmetic is binary so the normal form of 3.0 is actually $1.1_2 \times 2^1$. Since there are only two binary digits (1 and 0), by choosing that the first digit is not zero we have actually exactly chosen it to be 1. Because we know that the first digit will be a 1, we don't actually have to store it. We can "steal" the extra digit. Suppose we have a very small floating point number with 1 sign bit, 4 bits of mantissa and 3 bits of exponent. For simplicity we'll assume the bias is 0. Standard floating point numbers have the sign bit first, followed by the exponent and then the mantissa. So the number 3.0 will look like:

s	e	e	e	1	m	m	m	m
$\pm$	2^2	2^1	2^0	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
0	0	1	0	1	1	0	0	0

Recall that we don't actually store the 1 bit so the number will appear as 00101000.

This isn't actually a very useful floating point number format, though, because we have no capacity to store negative exponents so we can't store any numbers with absolute value less than 1. Our exponent can take values from 1 to 6[1]. We therefore choose a bias of 3 with the result that our effective exponent runs from -2 to +3. In this representation, the number 3 becomes:

s	e	e	e	1	m	m	m	m
$\pm$	2^2	2^1	2^0	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
0	1	0	0	1	1	0	0	0

The binary representation of 3.0 is therefore 01001000.

Exercise 2.1. Find the binary representation of the following numbers in a floating point format with one sign bit, a 3 bit exponent, a 4 bit mantissa and a bias of 3:

1. 1.0
2. 0.5
3. 10

2.1.1 Round-off error

Because floating point numbers have only finite precision, there are always numbers which can't be represented. In particular, it is not true that an arithmetic operation on two floating point numbers, if calculated exactly, will always result in a number which can be represented exactly. We are already familiar with this concept from decimal notation. For example there is no exact finite precision decimal representation of $\frac{1}{3}$.

By the definition of floating point numbers, the relative distance between adjacent representable numbers is fixed. Indeed, the largest *relative* gap in floating point numbers is given by the difference between 1 and the next floating point number greater than 1. In the case of our 8 bit floating point representation, the next number greater than 1 is given by $1 + 2^{-4}$ so the relative gap is $2^{-4} = 0.0625$. More generally, if we have an n_m bit mantissa, then the relative gap is 2^{-n_m}. This means that for every floating point number x, there is another floating point number y for which:

$$\frac{|x - y|}{|x|} \leqslant 2^{-n_m} \tag{2.2}$$

Another way of saying this is that for every real number x which lies within the range of our floating point numbers, there is a floating point number x' which lies not more than half this distance away:

$$|x - x'| \leqslant \frac{1}{2} 2^{-n_m} |x| \tag{2.3}$$

[1] an exponent value of zero or all ones has special meaning: see section 2.1.5

This quantity has a special name, *machine epsilon*:

$$\epsilon_{\text{machine}} = \frac{1}{2}2^{-n_m} \tag{2.4}$$
$$= 2^{-n_m-1} \tag{2.5}$$

The difference between a real number and its nearest floating point equivalent is known as round-off error. The relative round-off error is guaranteed to be no more than $\epsilon_{\text{machine}}$.

2.1.2 IEEE standard floating point numbers

The floating point numbers in use on essentially all modern computers employ many more bits than our simple 8. The Institute of Electronic and Electrical Engineers standards define various sizes:

precision	total size	sign bits	mantissa bits	exponent bits	exponent bias
half	16	1	11	5	15
single	32	1	23	8	127
double	64	1	52	11	1023
quad	128	1	112	15	1383

These values do not include the implicit 1 bit in the mantissa. In Python, the default is double precision and it is exceptionally rare to use any other size. For applications written in languages other than Python, single precision is often used. Quad precision is reserved for special applications and tends to be slow as hardware tends to only natively support single and double precision. Half precision floating point numbers are essentially never used in modern practice.

Exercise 2.2. Calculate $\epsilon_{\text{machine}}$ for IEEE single and double precision floating point numbers.

Exercise 2.3. Using the value of machine epsilon for double precision numbers, use Python to calculate `1.0+eps` and `1.0+2*eps`. What is the answer?

2.1.3 Floating point arithmetic

The four basic operations of arithmetic all have floating point equivalents, which, when it is necessary to make a distinction, are usually written: $\oplus$, $\ominus$, $\otimes$, and $\oslash$. If we write $\odot$ for any floating point operation and $\cdot$ for the corresponding exact operation, then the fundamental rule of floating point arithmetic is that for any floating point numbers x and y, there exists some ϵ with $|\epsilon| \leqslant \epsilon_{\text{machine}}$ such that:

$$x \odot y = (1+\epsilon)(x \cdot y) \tag{2.6}$$

In other words, the largest relative error which can occur in a single floating point operation is $\epsilon_{\text{machine}}$.

2.1.4 Catastrophic cancellation

A relative error of $\epsilon_{\text{machine}}$ might seem acceptable for all practical purposes, especially for double precision floats as $\epsilon_{\text{machine}}$ is really rather small. However, a particular problem occurs when adding and subtracting numbers which are very close together. Suppose we have some number x and a much smaller number δ and we calculate:

$$y = x + \delta \tag{2.7}$$
$$z = y - x \tag{2.8}$$

Clearly in exact arithmetic, $z = \delta$. What happens in floating point?

$$\begin{aligned} y &= (1+\epsilon_1)(x+\delta) \\ &= x + \delta + \epsilon_1 x + \epsilon_1 \delta \end{aligned} \tag{2.9}$$

for some $|\epsilon_1| < \epsilon_{\text{machine}}$. Then for some $|\epsilon_2| < \epsilon_{\text{machine}}$:

$$\begin{aligned} z &= y - x \\ &= (1+\epsilon_2)(\delta + \epsilon_1 x + \epsilon_1 \delta) \\ &= \delta + \epsilon_1 x + \epsilon_1 \delta + \epsilon_2 \delta + \epsilon_2 \epsilon_1 x + \epsilon_2 \epsilon_1 \delta. \end{aligned} \tag{2.10}$$

Most of the terms are guaranteed to be very small compared with the correct answer of δ but x is much larger than δ so $\epsilon_1 x$ may actually be significant. For sufficiently unfortunate input values and especially for repeated additions and subtractions of numbers very close together, the answer can end up with no correct digits!

Exercise 2.4. The expression $(1-x)^{10}$ can be expanded as

$$1 - 10x + 45x^2 - 120x^3 + 210x^4 - 252x^5 + 210x^6 - 120x^7 + 45x^8 - 10x^9 + x^{10}.$$

Use Python to evaluate this expression using the bracket form and then the expanded formula for the values 0.91, 0.92, 0.93, 0.94, 0.95 and 0.96. Which formula is more accurate? Why? *Hint:* how can you tell you have the correct answer?

2.1.5 Special values: 0, Inf and NaN*

The astute reader will have noticed a problem with the floating point format described here. If the first digit of the mantissa is always implicitly 1, how do we represent the number 0.? The answer is that the value 0 in the exponent field is treated as special. For our 8 bit floating point format, 0. is represented by:

s	e	e	e	1	m	m	m	m
$\pm$	2^2	2^1	2^0	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
0	0	0	0	1	0	0	0	0

so the actual stored value is 00000000. There is also a separate -0. value:

s	e	e	e	1	m	m	m	m
$\pm$	2^2	2^1	2^0	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
1	0	0	0	1	0	0	0	0

or 10000000. Positive and negative zero obey the usual sign rules for multiplication and division but compare equal. That is `1.0*-0.0` returns `-0.0` but `0.0==-0.0` is `True`.

Just as for integers, the problem of overflow occurs for floats: for any given size of floating point number, there are values too large to be represented. A related issue occurs if division by zero or another undefined operation occurs. For plain Python floating point numbers, the response in each of these cases is that an exception occurs:

```
In [97]: 10.**400
---------------------------------------------------------------
OverflowError                 Traceback (most recent call last)

/home/dham/<ipython console> in <module>()

OverflowError: (34, 'Numerical␣result␣out␣of␣range')

In [98]: 0./0.
---------------------------------------------------------------
ZeroDivisionError             Traceback (most recent call last)

/home/dham/<ipython console> in <module>()

ZeroDivisionError: float division
```

However as was the case with integer overflow, if we use a numeric Python array then we receive the floating point answer direct from the hardware:

```
In [101]: 0.0/numpy.array([0.0])
Out[101]: numpy.array([ NaN])

In [102]: numpy.array([10.])**400
Out[102]: numpy.array([ Inf])
```

These are two more special floating point values. `Inf` is short for infinity but should really be thought of as "a value too big to be represented". `Inf` is a signed quantity and follows the usual rules for positive and negative numbers so, for example `-1./numpy.array([0.])==numpy.array([-Inf])`. `NaN` means "not a number" and is returned in circumstances where even the sign of the answer is unknown. Some of the properties of the special floating point numbers are summarised in the following tables. In each case, x is an arbitrary finite positive floating point number. For each table, the row label comes before the operator and the column label after.

==	x	$-x$	0.0	−0.0	Inf	-Inf	NaN
x	T	F	F	F	F	F	F
$-x$	F	T	F	F	F	F	F
0.0	F	F	T	F	F	F	F
−0.0	F	F	F	T	F	F	F
Inf	F	F	F	F	T	F	F
-Inf	F	F	F	F	F	T	F
NaN	F	F	F	F	F	F	**F**

Notice that `NaN` compares false to itself!

$>$	x	$-x$	0.0	-0.0	`Inf`	`-Inf`	`NaN`
x	F	T	T	T	F	T	F
$-x$	F	F	F	F	F	T	F
0.0	F	T	F	F	F	T	F
-0.0	F	T	F	F	F	T	F
`Inf`	T	T	T	T	F	T	F
`-Inf`	F	F	F	F	F	F	F
`NaN`	F	F	F	F	F	F	F

Once again, notice that `NaN` always compares false.

$+$	x	$-x$	0.0	-0.0	`Inf`	`-Inf`	`NaN`
x	$2x$	0.0	x	x	`Inf`	`-Inf`	`NaN`
$-x$	0.0	$-2x$	$-x$	$-x$	`Inf`	`-Inf`	`NaN`
0.0	x	$-x$	0.0	0.0	`Inf`	`-Inf`	`NaN`
-0.0	x	$-x$	0.0	-0.0	`Inf`	`-Inf`	`NaN`
`Inf`	`Inf`	`Inf`	`Inf`	`Inf`	`Inf`	`NaN`	`NaN`
`-Inf`	`-Inf`	`-Inf`	`-Inf`	`-Inf`	`NaN`	`-Inf`	`NaN`
`NaN`	`NaN`	`NaN`	`NaN`	`NaN`	`NaN`	`NaN`	`NaN`

$\times$	x	$-x$	0.0	-0.0	`Inf`	`-Inf`	`NaN`
x	x^2	$-x^2$	0.0	-0.0	`Inf`	`-Inf`	`NaN`
$-x$	$-x^2$	x^2	-0.0	0.0	`-Inf`	`Inf`	`NaN`
0.0	0.0	-0.0	0.0	-0.0	`NaN`	`NaN`	`NaN`
-0.0	-0.0	0.0	-0.0	0.0	`NaN`	`NaN`	`NaN`
`Inf`	`Inf`	`-Inf`	`NaN`	`NaN`	`Inf`	`-Inf`	`NaN`
`-Inf`	`-Inf`	`Inf`	`NaN`	`NaN`	`-Inf`	`Inf`	`NaN`
`NaN`	`NaN`	`NaN`	`NaN`	`NaN`	`NaN`	`NaN`	`NaN`

$\div$	x	$-x$	0.0	-0.0	`Inf`	`-Inf`	`NaN`
x	1.0	-1.0	`Inf`	`-Inf`	0.0	-0.0	`NaN`
$-x$	-1.0	1.0	`-Inf`	`Inf`	-0.0	0.0	`NaN`
0.0	0.0	-0.0	`NaN`	`NaN`	0.0	-0.0	`NaN`
-0.0	-0.0	0.0	`NaN`	`NaN`	-0.0	0.0	`NaN`
`Inf`	`Inf`	`-Inf`	`Inf`	`-Inf`	`NaN`	`NaN`	`NaN`
`-Inf`	`-Inf`	`Inf`	`-Inf`	`Inf`	`NaN`	`NaN`	`NaN`
`NaN`	`NaN`	`NaN`	`NaN`	`NaN`	`NaN`	`NaN`	`NaN`

Exercise 2.5. * Write a program in Python which produces each of the tables above. It will be easier to use ± 1.0 rather than $\pm x$.

2.2 Python representation of integers and floats

Python's very flexible typing system means that it is often not necessary to pay attention to whether a number is an integer or a float. However it helps to know what the rules are.

2.2.1 Literals

Any number written without a decimal point is automatically stored as an integer, any number with a decimal point is a float. So, for example, `1` is an integer but `1.` or `1.0` is a float.

It is also possible to enter floats in scientific notation using the letter `e` to separate the mantissa from the exponent. So, for example, `5.e2` means 5×10^2 or in other words `500.`. Note that the base of the exponent is 10, not 2.

2.2.2 Casting to type

It is also possible to use a function to force a value to be a float or an integer. For example `float(10)` is the floating point value `10.0`. The corresponding function `int` operates by truncating the fractional part towards zero so, for example `int(1.9)` is the integer `1` and `int(-1.9)` is the integer `-1`. Using a function to change the type of a value is known as *casting*.

2.2.3 Special values

A special case of casting occurs if for some reason we wish to produce the special values `Inf`, `-Inf` and `NaN`. These are created with the commands `float('Inf')`, `float('-Inf')` and `float('NaN')` respectively.

2.3 Complex numbers

Python also has inbuilt support for complex numbers. For this, see section 1.6 in Langtangen.

2.4 Further exercises*

You learnt in Maths Methods 1 that the exponential function can be represented as the series:

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} \tag{2.11}$$

Exercise 2.6. * Write a Python module `series.py` containing a function `exp_series(x,n)` which approximates $\exp x$ using the first n terms of the sum in equation 2.11. Check your work by ensuring that `exp_series(1.0,100)` returns the same result as `math.exp(1.0)`. *Hint:* Cast `x` to `float` to ensure your function does the right thing if the user enters an integer.

Exercise 2.7. * Evaluate e^1 using your function for values of `n` ranging from 1 to 10. Now do the same for e^{-5}. What goes wrong? *Hint:* look at section 2.1.4

Exercise 2.8. ** Using the identity:

$$e^{-x} = \frac{1}{e^x},$$

produce a modified function `exp_safe(x,n)` which behaves correctly for negative inputs. Include `exp_safe` in your `series` module.

2.5 Further reading

The seminal work on floating point arithmetic is Goldberg, D. 1991. *What every computer scientist should know about floating-point arithmetic*. ACM Comput. Surv. 23, 1 (Mar. 1991), 5-48. DOI=http://doi.acm.org/10.1145/103162.103163

Chapter 3

Function approximation and big $\mathcal{O}$ notation.

3.1 Approximating functions

Now that we have covered the basic arithmetic operations, we can move on to consider functions. For the purposes of this module, we will only be interested in functions of one variable. The multi-variable case is more complex but the same principles apply. So, suppose we have a function $y = f(x)$. This function contains infinite information, for it takes a value y for each of an infinite set of inputs x. On a finite computer, this is a problem. Faced with the same problem for real numbers, we adopted the floating point system in which real arithmetic is approximated by a finite set of numbers.

We can adopt a similar approach for functions. First, we assume that we are only interested in the values of a function over some finite interval $[a, b]$. For most scientific purposes this is a completely reasonable assumption. Next we divide the interval into n equal-sized intervals with width $h = (b - a)/n$. As a first approximation, we could take the value of the function over each interval to be the value which the function takes at the left-hand end of that interval. Figure 3.1 illustrates this approach for the function:

$$f(x) = \sin(x) \tag{3.1}$$

on the interval $[0, \pi]$. Remember that we wanted to approximate our function at infinitely many points by a function which only requires finite storage on a computer. This approximation achieves this: we only need to record the five x values corresponding to the ends of the intervals and the four $f(x)$ values.

3.1.1 Plotting piecewise constant approximations

We can write a very short Python function which will plot a piecewise approximation to given function over a given interval using a given number of equal sub-intervals:

```
def plot_constant(f, a, b, n):
```

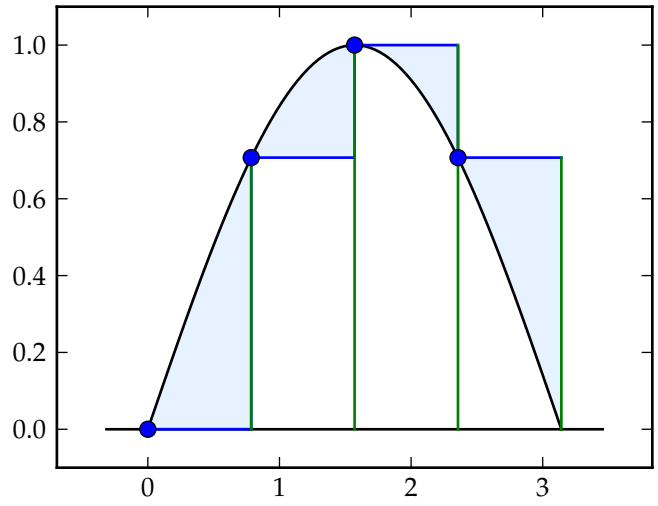

Figure 3.1: Piecewise constant approximation to the function $\sin(x)$ using four equal intervals. The approximate function is shown in blue, the actual function in black. The blue dots show the left-hand end of each interval and the shaded area indicates the error in the approximation.

```
    '''plot_constant(f, a, b, n)

    Plot a piecewise constant approximation to f over the
    interval [a,b] using n equal width intervals.'''
    # Import the plotting system.
    import pylab

    # Calculate the width of one interval.
    # Cast (b-a) to float to prevent integer division.
    h=float(b-a)/n

    # Create a new figure.
    fig=pylab.figure()

    # Plot each interval.
    for i in range(n):
        x=(h*i, h*(i+1))
        y=(f(x[0]), f(x[0]))
        pylab.plot(x,y,'b')

    # Expand the margins by 10% in each direction so we can see
    # the edge lines.
    pylab.margins(0.1)
```

The function is available in `/numerical-methods-1/approximation.py`. Copy this file to your working directory and run the function by executing `ipython -pylab` on the command line and then typing:

```
In [1]: from approximation import *

In [2]: def f(x):
 ...:     return sin(x)
 ...:

In [3]: plot_constant(f, 0, pi, 4)
```

The `approximation` module also contains the function `plot_constant_error`, which produces plots similar to figure 3.1.

Exercise 3.1. Write a short Python script named `plot_x_squared.py` which uses `approximation.plot_constant_error` to produce plots of approximations to the function x^2 on the interval $[0, 1]$ with $n = 2, 3, 4, 5$. Your script should be run by running `ipython -pylab` and then typing:

```
execfile("plot_x_squared.py")
```

Hint: your script could have the following form:

1. import the approximation module.
2. define a function which returns x^2.
3. loop over the list [2,3,4,5] calling `approximation.plot_constant_error` to produce the actual plots.

You should submit `plot_x_squared.py` online and in hard copy. There is no need to submit printouts of the plots.

3.2 Better approximations

Figure 3.1 is actually quite a bad approximation to $\sin(x)$. We can see this both because our piecewise constant approximation doesn't look much like a sine curve and also because the error, the area in blue, is large compared with the area under the curve. So what can we do about this? The first answer is that we can use more intervals: the smaller the intervals, the smaller the error. The second answer is that we can use a better approximation to the line than a series of horizontal lines. The approximation which is one step up from a horizontal line is to take the value at *each* end of each interval and draw a sloped line between them. Figure 3.2 shows approximations to $\sin(x)$ using piecewise constant and piecewise linear functions with various numbers of sub-intervals.

Two things are apparent by observation. The first is that the greater the number of intervals, the smaller the visible error. The second is that the piecewise linear approximation has much lower error than the piecewise constant approximation. We can make this a bit more formal by actually writing down the error we have made.

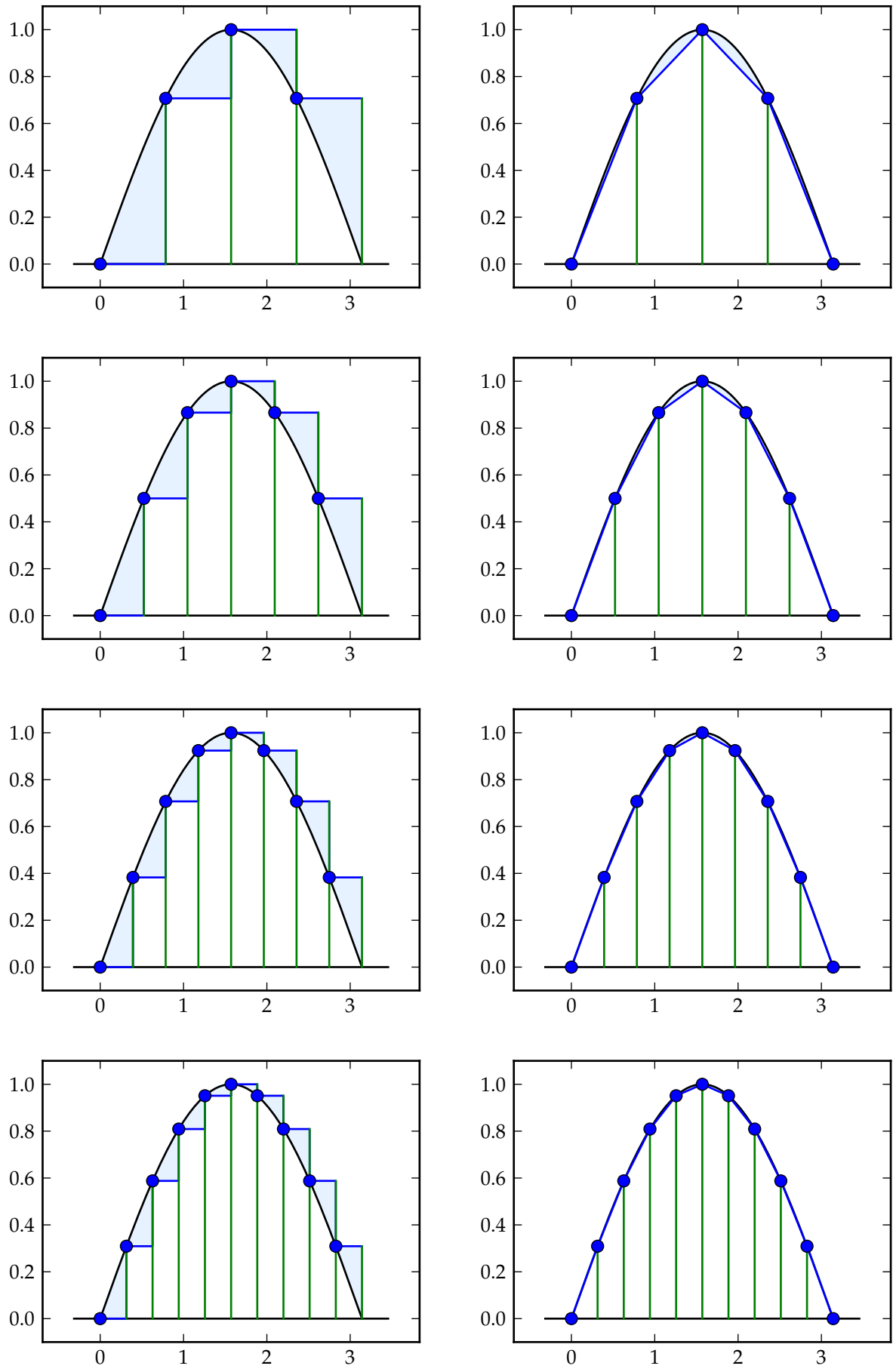

Figure 3.2: Piecewise constant (left) and piecewise linear (right) approximations to the function $\sin(x)$. Notice that the error is far lower for a given number of intervals in the linear approximation.

The error is the area between the actual function and our approximate function[1]. If we write $\hat{f}(x)$ for the approximate function, then we know from secondary school calculus that:

$$E = \int_a^b |f(x) - \hat{f}(x)| \, \mathrm{d}x \tag{3.2}$$

To actually calculate this integral, we can split it into a sum of integrals, one over each sub-interval:

$$E = \sum_{i=0}^{n-1} \int_{a+hi}^{a+h(i+1)} |f(x) - \hat{f}(x)| \, \mathrm{d}x \qquad h = \frac{b-a}{n} \tag{3.3}$$

Note that $x = a + ih$ is the left hand end of the i-th interval while $x = a + (i+1)h$ is the right hand end of that interval.

This makes our life a little easier since we can write an explicit expression for $\hat{f}$ on each sub-interval since it's just constant:

$$\hat{f}(x) = f(ih), \qquad a + ih \leqslant x < a + (i+1)h \tag{3.4}$$

We could now substitute in $f(x) = \sin(x)$ and attempt to work out this integral by hand, but actually it's easier to just use a Python routine provided the `scipy.integrate` module to integrate this expression numerically:

```
def constant_error(f, a, b, n):
  '''Return the integral of the error when f is approximated
  over the interval [a,b] by n equal width piecewise constant
  functions.'''
  import scipy.integrate as si

  error=0
  h=float(b-a)/n

  for i in range(n):
      l=a+i*h
      r=a+(i+1)*h

      def f_err(x):
        return abs(f(x)-f(l))

      # si.quad is the numerical integration routine.
      error+=si.quad(f_err,l,r)[0]

  return error
```

We can do the same calculation for the error in the piecewise linear case. This time we need to use the equation for a straight line $y = y_0 + m(x - x_0)$ where m is the gradient

[1]There are different sorts of error we could measure. For example we might be interested in the error in $f(x_0)$ for a given fixed x_0. The error we're using here measures the error over the whole interval $[a, b]$, we'll meet some other errors later.

of our line and (x_0, y_0) is our starting point. Our starting point is $(a+ih, f(a+ih))$ and we can calculate the gradient of a straight line using its two end points:

$$\hat{f}(x) = f(a+ih) + (x-(a+ih))\frac{f(a+(i+1)h) - f(a+ih)}{h}, \quad a+ih \leqslant x < a+(i+1)h \tag{3.5}$$

This yields a very similar error function for the piecewise linear case:

```
def linear_error(f, a, b, n):
  '''Return the integral of the error when f is approximated
  over the interval [a,b] by n equal width piecewise linear
  functions.'''
  import scipy.integrate as si
  error=0
  h=float(b-a)/n

  for i in range(n):
      l=a+i*h
      r=a+(i+1)*h
      dy=f(r)-f(l)
      def f_err(x):
          x_bar=(x-l)
          return abs(f(x)-(f(l)+x_bar*dy/h))

      error+=si.quad(f_err,l,r)[0]

  return error
```

By evaluating those two functions for various numbers of sub-intervals, we can calculate how the error in each approximation decreases as the interval width falls. Figure 3.3 shows this for our usual example of $\sin(x)$.

Figure 3.3 accords with our observations that error falls with h and that the piecewise linear approximation has much lower error for a given h than does the piecewise constant approximation. However something even more interesting is apparent if we look at the *shape* of each of the two curves. As h becomes small, the error in the piecewise constant starts to look like a straight line. For small h, the piecewise linear approximation is curved, perhaps more like a quadratic function.

Understanding this behaviour is very important, because in general we do *not* know an expression for the function we are approximating or the value of the error. However if we know that the error is linear in h, then if we double out number of sub-intervals then the error will halve. If we know that the error is quadratic in h then doubling the number of sub-intervals will cause the error to reduce by a factor of 4, which is much better.

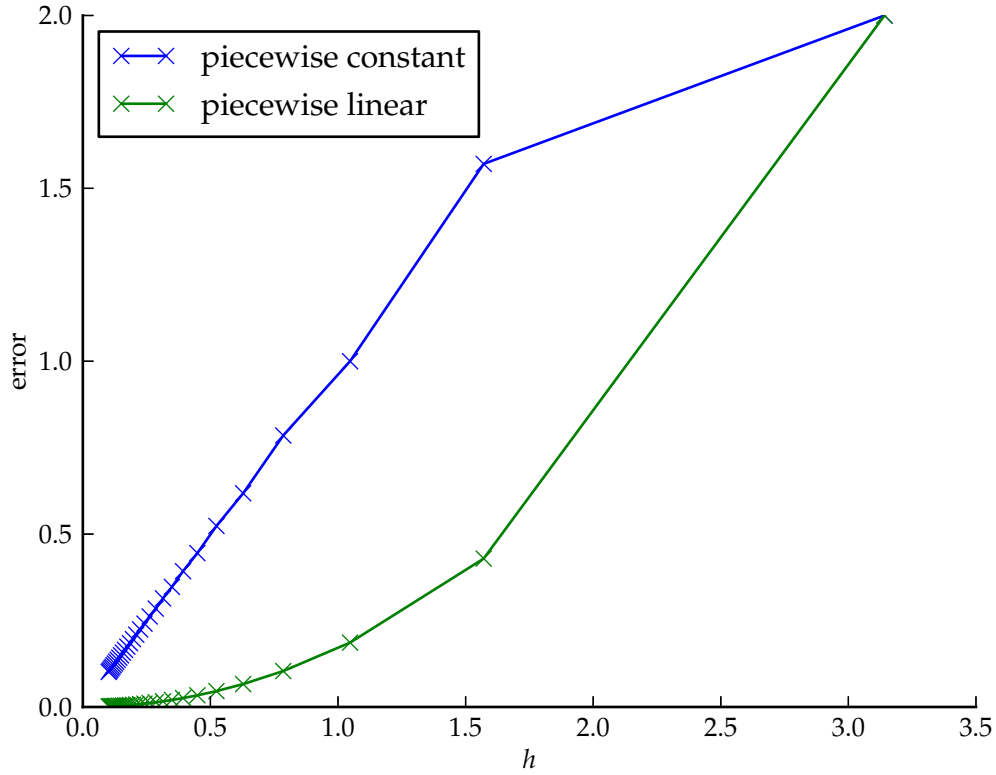

Figure 3.3: Error in piecewise constant and piecewise linear approximations of $\sin(x)$ on the interval $[0, \pi]$ as a function of sub-interval width h.

3.3 Big $\mathcal{O}$ notation

Because the study of the scaling of functions is so important, there is a special set of notation which has grown up to express it.

Definition 3.1. A function $f(x)$ is $\mathcal{O}(\alpha(x))$ as $x \to a$ if there exists some positive constant c such that for any choice of x sufficiently close to a:

$$|f(x)| < c\alpha(x)$$

Most often, α will be x^n for some power n.

Big $\mathcal{O}$ notation gives an *upper bound* on the *magnitude* of a function. This is useful in two important contexts, we want to know that the error in a calculation gets smaller at a particular rate as we use smaller steps. We also often want to know that the amount of work, for example the number of floating point operations, will not increase faster than a particular rate as we increase the problem size. At the moment we're concerned with the former of these: showing that the error is small.

We're interested in particular in the case were the error $E(h)$ is $\mathcal{O}(h^n)$ as $h \to 0$. One critical result to notice is that h^n approaches zero faster for higher values of n. The symbol $\mathcal{O}$ is usually pronounced "order", so if f is $\mathcal{O}(h^2)$, we would say "f is order h squared".

The following program plots x^n for increasing n demonstrating this point:

```
import pylab
import time

# 101 evenly spaced points in [0,1]
x=pylab.linspace(0,1,101)

for i in range(20):
  pylab.plot(x, x**(i+1))
  pylab.draw() # Force plot update.
  time.sleep(0.3)
```

To actually see the animation, run `ipython -pylab` and run this program in that context.

The consequence of this result is the following:

Theorem 3.2. If $E(h)$ is $\mathcal{O}(h^n)$ as $h \to 0$ then $E(h)$ is $\mathcal{O}(h^m)$ as $h \to 0$ for all $m > 0$ less than n.

When we talk about *the* order of convergence of a numerical method, we mean the maximum value of n for which $E(h)$ is $\mathcal{O}(h^n)$ as $h \to 0$. If we are talking about errors, we are always interested in the case where $h \to 0$ so it is usual to leave this out and just say $E(h)$ is $\mathcal{O}(h^n)$.

Another very important, if counterintuitive, result is the addition rule for $\mathcal{O}$:

Theorem 3.3. If $E(h)$ is $\mathcal{O}(h^n)$ and $F(h)$ is $\mathcal{O}(h^n)$ then $E(h) + F(h)$ is $\mathcal{O}(h^n)$.

Proof: By the definition of $\mathcal{O}$, we have:

$$|E(h)| < c_1 h^n \tag{3.6}$$
$$|F(h)| < c_2 h^n \tag{3.7}$$

for some $c_1 > 0$, some $c_2 > 0$ and for h sufficiently close to zero. Adding these two inequalities, we have:

$$|E(h)| + |F(h)| < (c_1 + c_2) h^n \tag{3.8}$$

But we also know[2] that:

$$|E(h) + F(h)| \leqslant |E(h)| + |F(h)| \tag{3.9}$$

So if we write $c_3 = c_1 + c_2$ then we have:

$$|E(h) + F(h)| < c_3 h^n \tag{3.10}$$

with $c_3 > 0$ for h sufficiently close to zero. Hence by the definition of $\mathcal{O}$, $E(h) + F(h)$ is $\mathcal{O}(h^n)$.

Q.E.D.

[2]This result is known as the triangle inequality. To see that it is true note that where $E(h)$ and $F(h)$ have the same sign, $|E(h) + F(h)| = |E(h)| + |F(h)|$. However where $E(h)$ and $F(h)$ different sign, they partially cancel so $|E(h) + F(h)| < |E(h)| + |F(h)|$

Exercise 3.2. Suppose that $E(h)$ is $\mathcal{O}(h^n)$. If we define:

$$F(h) = E(h)/h \tag{3.11}$$

prove that $F(h)$ is $\mathcal{O}(h^{n-1})$

3.3.1 Numerically estimating the order of convergence

Suppose our error $E(h)$ is actually converging at approximately the rate of h^n. Then we could write:

$$|E(h)| \approx ch^n \tag{3.12}$$

For some unknown constant c. Suppose we know the error in our method for two resolutions, h_1 and h_2, then we can write:

$$|E(h_1)| \approx ch_1^n \tag{3.13}$$
$$|E(h_2)| \approx ch_2^n. \tag{3.14}$$

Equivalently:

$$\left|\frac{E(h_1)}{E(h_2)}\right| \approx \left(\frac{h_1}{h_2}\right)^n \tag{3.15}$$

By taking logarithms, we can find an expression for n:

$$\ln\left(\left|\frac{E(h_1)}{E(h_2)}\right|\right) \approx n \ln\left(\frac{h_1}{h_2}\right) \tag{3.16}$$
$$n \approx \frac{\ln\left(\left|\frac{E(h_1)}{E(h_2)}\right|\right)}{\ln\left(\frac{h_1}{h_2}\right)} \tag{3.17}$$

Suppose now that we have a sequence of error values for decreasing h. If the order of convergence of our method is n then we would expect the right hand side of equation (3.17) to converge to n as we move along the sequence.

Turning this statement around, if we have a sequence of error values for decreasing h then we can estimate the order of convergence by evaluating equation (3.17) for each adjacent pair of h and E values.

Exercise 3.3. Write a Python module `convergence` containing a function `convergence(h,s)` which takes a sequence of h values and a matching sequence of error values s and returns a sequence of estimates of convergence orders using equation (3.17).

Copy `/numerical-methods-1/test_convergence.py` to your working directory. You can run the same test of your code that the demonstrators will use by running:

```
> python test_convergence.py
```

You can also test your function by hand by running `ipython` and typing:

```
import convergence
from test_convergence import S1,S2,h
convergence.convergence(h,S1)
convergence.convergence(h,S2)
```

In the first case, your `convergence` function should return a sequence which converges towards 1.0, in the second case your sequence should converge towards 2.0.

Chapter 4

Taylor series and numerical integration

4.1 Taylor series and error

From maths methods 2, you know that a smooth function can be approximated close to a point x_0 by a Taylor series, the formula for which is:

$$f(x_0 + h) = \sum_{k=0}^{\infty} \frac{h^k}{k!} f^{(k)}(x_0) \tag{4.1}$$

The derivation of the Taylor series is presented in appendix A. When we actually use a Taylor series to approximate a function, we can't evaluate the whole infinite series, instead we evaluate the first few terms. It is therefore interesting to investigate what sort of error we make by just summing a finite number of terms in the series.

Let's look at the k-th term of the Taylor series:

$$f_k = \frac{h^k}{k!} f^{(k)}(x_0) \tag{4.2}$$

The factor:

$$\frac{f^{(k)}(x_0)}{k!} \tag{4.3}$$

is just a constant value so we can replace this by some constant c_k so that:

$$f_k = c_k h^k \tag{4.4}$$

That is, f_k is $\mathcal{O}(h^k)$.

Suppose we evaluate n terms of the series. Using $\mathcal{O}$ notation for the remaining terms, we have:

$$f(x_0 + h) = \sum_{k=0}^{n-1} \frac{h^k}{k!} f^{(k)}(x_0) + \sum_{k=n}^{\infty} \mathcal{O}(h^k) \tag{4.5}$$

Next we can look back at theorem 3.3, the addition rule for $\mathcal{O}$. This tells us that if we add several functions which are $\mathcal{O}(h^k)$ for various values of k, the result will be

$\mathcal{O}(h^m)$ where m is the least of the k values[1]. In our case, the least order is n so we may write:

$$f(x_0 + h) = \sum_{k=0}^{n-1} \frac{h^k}{k!} f^{(k)}(x_0) + \mathcal{O}(h^n) \tag{4.6}$$

In other words, if we approximate a smooth function by the first n terms of a Taylor series, then the error in the value $f(x_0 + h)$ is $\mathcal{O}(h^n)$.

4.2 Adding $\mathcal{O}(h^n)$ error terms

In the preceding section we used expressions of the form:

$$f(x_0 + h) = \sum_{k=0}^{n-1} a_k h^k + \mathcal{O}(h^n) \tag{4.7}$$

This is a new use of the $\mathcal{O}$ notation: previously we merely used $\mathcal{O}$ as a property of a function in statements of the form "$f(h)$ is $\mathcal{O}(h^n)$". Adding $\mathcal{O}(h^n)$ to the end of an expression is a very common practice to indicate the size of an error term. The notation "$+\mathcal{O}(h^n)$" should be understood as meaning "plus an unknown function which is $\mathcal{O}(h^n)$".

This notation has a couple of possibly surprising properties. First, adding multiple O terms *never* cancels to zero. For example, suppose we have two equations:

$$f(h) = ah + bh^2 + \mathcal{O}(h^3) \tag{4.8}$$

$$g(h) = ch + dh^2 + \mathcal{O}(h^3) \tag{4.9}$$

If we subtract g from f then we have:

$$f(h) - g(h) = ah + bh^2 - ch - dh^2 + \mathcal{O}(h^3) \tag{4.10}$$

This is because each use of $\mathcal{O}$ is potentially a different unknown error function. Subtracting them from each other therefore does not result in cancellation. In accordance with the addition rule for $\mathcal{O}$ (theorem 3.3), if error terms with different powers of h are added, only the term with the *smallest* power need be kept.

Second, the $\mathcal{O}$ term, by convention, is always positive. The absolute value signs in the definition of $\mathcal{O}$ (definition 3.1) mean that the sign of a function does not change its $\mathcal{O}$ properties.

4.3 Taylor series and function approximation

The Taylor series is possibly the most important tool in numerical analysis. To demonstrate this, we can apply it to our function approximations from the previous chapter.

[1] In explaining this, we intentionally skip over sum of the subtleties of infinite sums. The result stated is correct in all cases where the radius of convergence of the Taylor series is not 0.

Let's start with our piecewise constant approximation. In the i-th interval, the expression for this approximation is:

$$\hat{f}(x) = f(a+ih) \quad a+ih \leqslant x < a+(i+1)h \tag{4.11}$$

Remember that:

$$h = \frac{b-a}{n} \tag{4.12}$$

where h is the sub-interval width, b is the right hand limit of the whole interval, a is the left hand limit and n is the number of sub-intervals.

In order to work out the error, we can write the original function as a Taylor series:

$$f(a+ih+\delta x) = f(a+ih) + \mathcal{O}(\delta x) \qquad \delta x = x-(a+ih) \tag{4.13}$$

Now we can write the error at a single point as:

$$\begin{aligned} E(x) &= \left| f(x) - \hat{f}(x) \right| \\ &= |f(a+ih) + \mathcal{O}(\delta x) - f(a+ih)| \\ &= |\mathcal{O}(\delta x)| \end{aligned} \tag{4.14}$$

Due to the arbitrary constant in the definition of $\mathcal{O}$, we can drop the absolute value signs and simpy say that the error is $\mathcal{O}(\delta x)$. That's not quite what we were after, though, because we really wanted an expression for the error in terms of the interval size h. Fortunately we know how to relate δx and h. Remember that for any x we can find the interval containing x and label it i. This implies

$$a+ih \leqslant x < a+(i+1)h \tag{4.15}$$

In equation 4.13 we then defined:

$$\delta x = x-(a+ih) \tag{4.16}$$

By combining these together we see that $\delta x < h$. From this it can be shown that the error in the piecewise constant approximation is $\mathcal{O}(h)$.

Exercise 4.1. If $F(h)$ is $\mathcal{O}(\delta x)$ and $0 < \delta x < h$, prove that F is $\mathcal{O}(h)$.

4.4 The error in the piecewise linear approximation

We can apply the same technique, in slightly more complex form, to derive the error in the piecewise linear approximation.

Assume we wish to evaluate the piecewise approximation $\hat{f}$ at a point x and that $a+ih < x < a+(i+1)h$. To simplify the notation a bit, we will introduce l and $l+h$ as the ends of the interval containing x. That is,

$$l = a+ih \tag{4.17}$$

$$l+h = a+(i+1)h \tag{4.18}$$

We'll also write $\delta x = x - l$ for the position of x in the interval. The piecewise linear approximation given in equation 3.5 therefore becomes:

$$\hat{f}(l + \delta x) = f(l) + \delta x \frac{f(l+h) - f(l)}{h} \tag{4.19}$$

Now, we can write $f(l+h)$ as a Taylor series centred on l:

$$f(l+h) = f(l) + hf'(l) + \mathcal{O}(h^2) \tag{4.20}$$

If we substitute this into equation 4.19, we have:

$$\begin{aligned} \hat{f}(l + \delta x) &= f(l) + \delta x \frac{f(l) + hf'(l) + \mathcal{O}(h^2) - f(l)}{h} \\ &= f(l) + \delta x f'(l) + \frac{\delta x}{h}\mathcal{O}(h^2) \end{aligned} \tag{4.21}$$

If we note that $0 < \delta x < h$ then we can conclude that the error term is still bounded by $\mathcal{O}(h^2)$:

$$\hat{f}(l + \delta x) = f(l) + \delta x f'(l) + \mathcal{O}(h^2) \tag{4.22}$$

This looks quite close to what we're after, but we want the error in $\hat{f}(l + \delta x)$ as compared with $f(l + \delta x)$, so we now write a Taylor series expansion for the original function $f(l + \delta x)$ centred at l:

$$f(l + \delta x) = f(l) + \delta x f'(l) + \mathcal{O}(h^2) \tag{4.23}$$

As before, we have used $0 < \delta x < h$ to write the error term as $\mathcal{O}(h^2)$ rather than $\mathcal{O}(\delta x^2)$. We can rearrange this to isolate the terms on the right hand side of equation 4.22:

$$f(l) + \delta x f'(l) = f(l + \delta x) + \mathcal{O}(h^2) \tag{4.24}$$

We can substitute this last equation into equation 4.22 in order to finally write:

$$\hat{f}(l + \delta x) = f(l + \delta x) + \mathcal{O}(h^2) \tag{4.25}$$

We can therefore conclude that the pointwise error in the piecewise linear approximation of a (smooth) function is $\mathcal{O}(h^2)$. Another way to say this is to describe the piecewise linear approximation as *second order*.

4.5 Types of error

In everyday parlance, the word "error" means a mistake, or something which is wrong. In science and in scientific computing in particular, error has a different meaning. In these contexts, "error" is the difference between a computed result and the "true" result. There are a number of forms of error, some of the most important of which are:

Rounding error is due to the finite precision of floating point arithmetic. It is the difference between the result of a floating point operation and the infinite precision result.

Truncation error is due to ignoring the higher order terms in an infinite series, such as a Taylor series.

Model error is due to the approximations made in our description of the real world. For example, if we model a pendulum neglecting friction in the fulcrum and air resistance, these are model errors.

Error is an inevitable part of science but, as we saw with the catastrophic cancellation examples, it is essential to remain aware of error in calculations lest it come to dominate the actual result!

4.6 Numerical Integration

In section 3.2, we were interested in the error integrated over the whole interval $[a, b]$. This is particularly relevant for one very important use of numerical mathematics: integrating functions which we don't know how to integrate analytically. Let's first consider the piecewise linear approximation to a function. On a single interval, $[l, l + h]$, we can use equation 4.25 to write:

$$\int_l^{l+h} \hat{f}(x)\mathrm{d}x = \int_l^{l+h} f(x)\mathrm{d}x + \int_l^{l+h} \mathcal{O}(h^2)\mathrm{d}x \tag{4.26}$$

We understand the first two terms of this equation: the integral of the piecewise linear approximation is equal to the integral of the original equation plus an error term. But what happens when we integrate $\mathcal{O}(h^2)$? Remember, the notation $\mathcal{O}(h^2)$ when used as an error term, means an unknown function $E(x)$ which is $\mathcal{O}(h^2)$. The integral of the error function is bounded by the maximum magnitude of that function multiplied by the width of the interval over which integration occurs. That is:

$$\int_l^{l+h} E(x)\mathrm{d}x < h\left(\max_{l<x<x+h} E(x)\right) \tag{4.27}$$

Figure 4.1 illustrates this bound. The blue shaded area is the true error on the left of the inequality, while the right of the inequality is the sum of the blue and pink areas.

We know $E(x)$ is $O(h^2)$ for any x in $[l, l + h]$ so in particular, it is $O(h^2)$ at its maximum. This means that the integral error is $h\mathcal{O}(h^2)$. By inverting the division rule for $\mathcal{O}$ (exercise 3.2) this yields the result that the error is $\mathcal{O}(h^3)$.

4.6.1 Evaluating the trapezoidal rule

The shape of the approximate integration area shown in figure 4.1 gives this form of numerical integration its name: the trapezoidal rule. To actually evaluate the integral, we can write out the definition of $\hat{f}$ and integrate it. Using equation 4.19, we have:

$$\int_l^{l+h} \hat{f}(x)\mathrm{d}x = \int_0^h f(l) + \delta x \frac{f(l+h) - f(l)}{h}\mathrm{d}\delta x \tag{4.28}$$

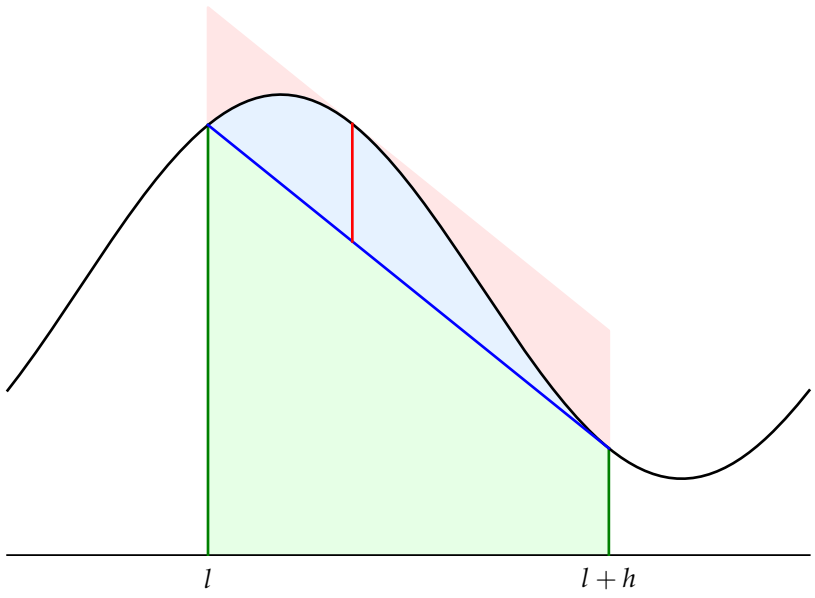

Figure 4.1: Integration of the piecewise linear approximation to a function over one sub-interval $[l, l+h]$. The approximate integral is shown in green and the error in blue. The maximum pointwise error is shown by a red line and the bound on the integral error is given by the pink area plus the blue area. It is clear that the sum of the pink area and the blue area is greater than the blue area on its own.

Note that we have changed variable to δx in the right hand side integral. This is a very easy integral to evaluate:

$$\begin{aligned}\int_l^{l+h} \hat{f}(x)\mathrm{d}x &= hf(l) + \frac{h^2}{2}\frac{f(l+h)-f(l)}{h} \\ &= hf(l) + \frac{h}{2}\left(f(l+h)-f(l)\right) \\ &= \frac{h}{2}f(l) + \frac{h}{2}f(l+h)\end{aligned} \tag{4.29}$$

We can combine this with the error result to write a single statement for the trapezoidal rule and its error:

$$\int_l^{l+h} f(x) = \frac{h}{2}f(l) + \frac{h}{2}f(l+h) + \mathcal{O}(h^3) \tag{4.30}$$

4.6.2 The multi-interval trapezoidal rule

The formula above is a useful integration rule for a single sub-interval, but we really need to integrate over $[a, b]$ divided into any number of sub-intervals. Fortunately

this is easily achieved by applying equation 4.30 to each sub-interval:

$$\int_a^b f(x)\mathrm{d}x = \sum_{i=0}^{n-1}\left(\frac{h}{2}f(a+ih)+\frac{h}{2}f(a+(i+1)h)+\mathcal{O}(h^3)\right) \quad n=\frac{b-a}{h} \tag{4.31}$$

This gives us a rule for calculating the integral, but what happens to the error? We sum the n error terms, each of which is $\mathcal{O}(h^3)$. If we sum a fixed number of $\mathcal{O}(h^3)$ error terms then we already know that the result will be $\mathcal{O}(h^3)$. However, n isn't fixed, it's a function of h so as h becomes small, n becomes ever larger. Let's write $E_i(h)$ as the i-th error term. We can once again bound the error using the largest error term:

$$\sum_{i=0}^{n-1} E_i(h) < n\left(\max_{0\leqslant i<n}|E_i(h)|\right) = \frac{a-b}{h}\left(\max_{0\leqslant i<n}|E_i(h)|\right) \tag{4.32}$$

Now, $E_i(h)$ is $\mathcal{O}(h^3)$ and we know from exercise 3.2 that dividing an $\mathcal{O}(h^3)$ by h produces an $\mathcal{O}(h^2)$ function. $b-a$ is the length of the whole interval and doesn't change when we change the number of sub-intervals, so it will be absorbed in the arbitrary constant, c, in the definition of $\mathcal{O}$. We can therefore conclude that the overall error in the multi-interval trapezoidal rule is $\mathcal{O}(h^2)$:

$$\int_a^b f(x)\mathrm{d}x = \sum_{i=0}^{n-1}\left(\frac{h}{2}f(a+ih)+\frac{h}{2}f(a+(i+1)h)\right)+\mathcal{O}(h^2) \quad n=\frac{b-a}{h} \tag{4.33}$$

The analysis above does not depend on the particular characteristics of the trapezoidal rule: it is a general rule that the multi-interval error term will be one order lower than that for the single interval. The general term form the single interval error is the *local truncation error* while multi-interval term is the *global truncation error*.

4.6.3 Simpson's rule

We can obtain a still more accurate representation of the integral of a function, f, over a single sub-interval $[l, l+h]$ by approximating f with a quadratic function. Figure 4.2 illustrates this approximation.

Since a quadratic function is defined by the function value at three points (as opposed to two for a linear function and one for a constant function), Simpson's rule employs the value of the function at the centre of the interval as well as the end points. We will state without proof that the formula for Simpson's rule on a single interval is:

$$\int_l^{l+h} f(x)\mathrm{d}x = h\left(\frac{1}{6}f(l)+\frac{2}{3}f\left(l+\frac{1}{2}h\right)+\frac{1}{6}f(l+h)\right)+\mathcal{O}(h^5) \tag{4.34}$$

Using the same argument as above, the formula for Simpson's rule over multiple intervals is:

$$\int_a^b f(x)\mathrm{d}x = \sum_{i=0}^{n-1} h\left(\frac{1}{6}f(a+ih)+\frac{2}{3}f\left(a+\left(i+\frac{1}{2}\right)h\right)+\frac{h}{6}f(a+(i+1)h)\right)+\mathcal{O}(h^4)$$
$$n=\frac{b-a}{h} \tag{4.35}$$

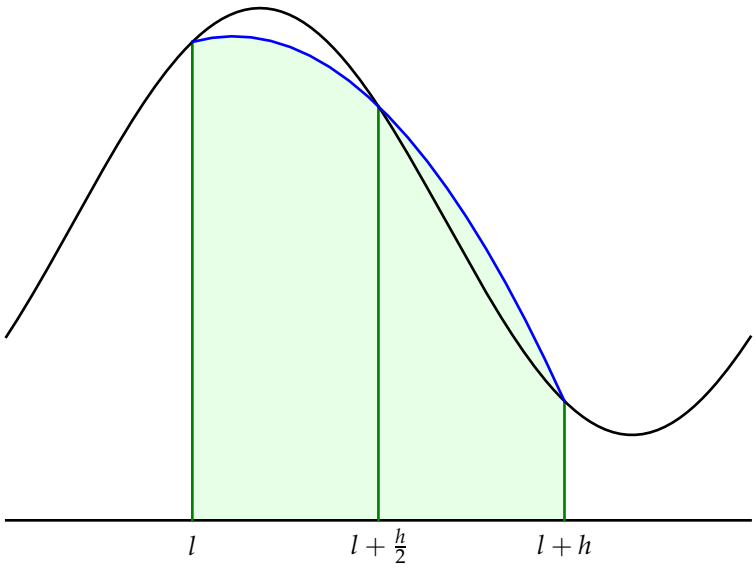

Figure 4.2: Simpson's rule approximates the integral of $f(x)$ on the interval $[l, l+h]$ using a qudratic function.

4.7 Numerical Quadrature

The trapezoidal rule and Simpson's rule are two specific examples of the general system of numerical integration known as numerical quadrature. The basic principle of numerical quadrature is that the integral of a function can be calculated by evaluating that function at a particular set of points and computing a weighted sum of those values. The general form of a quadrature rule is a set of quadrature points $\{x_j | j = 0 \ldots q-1\}$ in the interval $[0,1]$ and a set of corresponding weights $\{w_j | j = 0 \ldots q-1\}$. The integral on a single sub-interval $[l, l+h]$ is then approximated using the formula:

$$\int_l^{l+h} f(x)\mathrm{d}x \approx \sum_{j=0}^{q-1} w_j h f(l + x_j h) \tag{4.36}$$

In order to calculate the integral over the full interval $[a, b]$, this procedure is repeated for each sub-interval:

$$\int_a^b f(x)\mathrm{d}x \approx \sum_{i=0}^{n-1} \sum_{j=0}^{q-1} w_j h f(a + ih + x_j h) \tag{4.37}$$

The trapezoidal rule and Simpson's rule are two members of a more general family known as the Newton-Cotes formulae. Table 4.1 shows three of its members.

Some patterns are apparent in these quadrature rules. The first is that the quadrature points are equally spaced in the interval $[0,1]$. This is characteristic of the Newton-

Name	Quadrature points (x_j)	Weights (w_j)	Global truncation error
trapezoidal rule	$[0, 1]$	$\left[\frac{1}{2}, \frac{1}{2}\right]$	$\mathcal{O}(h^2)$
Simpson's rule	$\left[0, \frac{1}{2}, 1\right]$	$\left[\frac{1}{6}, \frac{2}{3}, \frac{1}{6}\right]$	$\mathcal{O}(h^4)$
Boole's rule	$\left[0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\right]$	$\left[\frac{7}{90}, \frac{16}{45}, \frac{6}{45}, \frac{16}{45}, \frac{7}{90}\right]$	$\mathcal{O}(h^6)$

Table 4.1: Newton-Cotes quadrature rules. This form of the rules is for h in the interval $[0, 1]$. Other sources may use other defintions of h with appropriate rescaling of the weights.

Cotes quadrature rules but is not true for all families of quadrature. For instance, the widely used Gauss-Lobatto quadrature rules employ unevenly spaced points. The second pattern to notice is that the quadrature weights always sum to 1. This is a general pattern which has to be true for all quadrature formulae on the interval $[0, 1]$. To see why this is so, consider the constant function $f(x) = 1$. A quadrature rule which can not even evaluate the integral of this function would be rather useless.

A further important pattern is that more accurate quadrature rules generally require the function to be evaluated at more quadrature points.

The file `/numerical-methods-1/newton_cotes.py` contains Python code for these quadrature rules. The module contains a dictionary `newton_cotes` containing the three quadrature rules from table 4.1 under the keys "trapezoidal", "Simpson" and "Boole" respectively. Each quadrature rule is represented by an object of class `Quadrature` containing a member `x`, which is the array of quadrature points and `w`, the array of weights. This can be accessed as follows:

```
In [1]: import newton_cotes

In [2]: q=newton_cotes.newton_cotes["Simpson"]

In [3]: q.x
Out[3]: array([ 0. ,  0.5,  1. ])

In [4]: q.w
Out[4]: array([ 0.16666667,  0.66666667,  0.16666667])
```

Exercise 4.2. Write a Python module named `quadrature` containing a function `quad(f, a, b, n, rule)`. `quadrature.quad` should calculate:

$$\int_a^b f(x)\mathrm{d}x$$

numerically using equation 4.37 over `n` sub-intervals. The argument `rule` should be a string containing "trapezoidal", "Simpson" or "Boole" and you should use the corresponding quadrature points and weights from the `newton_cotes` module. If `rule` is not one of the known quadrature rules, your function should raise `ValueError` with an appropriate error message.

You can use the script `test_quadrature.py` from `/numerical-methods-1` to test your function by typing:

```
> python test_quadrature.py
```

For this test to work, `test_quadrature.py`, `quadrature.py` and `newton_cotes.py` must all be in the same directory.

4.8 Quadrature in Python

The `scipy.integrate` module contains a very sophisiticated quadrature function `quad` which will evaluate integrals numerically to a very high precision. For a simple case, we can try integrating $f(x) = x\tan(x)$ on the interval $[0,1]$:

```
In [1]: from numpy import tan

In [2]: from scipy.integrate import quad

In [3]: def f(x):
   ...:     return x*tan(x)
   ...:

In [4]: quad(f,0,1)
Out[4]: (0.42808830136517606, 1.1548411421164902e-14)
```

The first number in the output is the answer, the second is an error estimate.

Exercise 4.3. $f(x) = x\tan(x)$ is a very simple example of a function with no simple integral. Write a python script `integrate_x_tan.py` which uses your `quadrature.quad` function to integrate this function on the interval $[0,1]$ using each of the quadrature rules above and for $n = 2,4,8,16$. Using your `convergence` function from exercise 3.3, check that the integral converges at the expected rate from table 4.1 for each quadrature rule. Use `scipy.integrate.quad` to generate a "true" answer in order to compute the error for your convergence calculations. Your script should print out the convergence rate estimates along with suitable documentation.

Chapter 5

Numerical Differentiation

5.1 Forward differencing

Just as numerical quadrature enables us to integrate functions by simply evaluating the function at a number of points, we can also calculate approximations to the derivatives of a function using weighted sums of function evaluations.

Remember the elementary definition of the derivative of a function f at a point x_0:

$$f'(x_0) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{5.1}$$

We can turn this into an approximation rule for $f'(x)$ by replacing the limit as h approaches 0 with a small but finite h:

$$f'(x_0) \approx \frac{f(x_0+h) - f(x)}{h}, \qquad h > 0 \tag{5.2}$$

Figure 5.1 illustrates this approximation. Because the approximate gradient is calculated using values of x greater than x_0, this algorithm is known as the *forward difference method*.

So how accurate is this method? For this we can fall back on our old friend the Taylor series. We can write:

$$f(x_0+h) = f(x_0) + hf'(x_0) + \mathcal{O}(h^2) \tag{5.3}$$

If we rearrange this expression to isolate the gradient term $f'(x_0)$ on the left hand side, we find:

$$hf'(x_0) = f(x_0+h) - f(x_0) + \mathcal{O}(h^2) \tag{5.4}$$

$$f'(x_0) = \frac{f(x_0+h) - f(x_0)}{h} + \mathcal{O}(h) \tag{5.5}$$

So the forward difference method is first order.

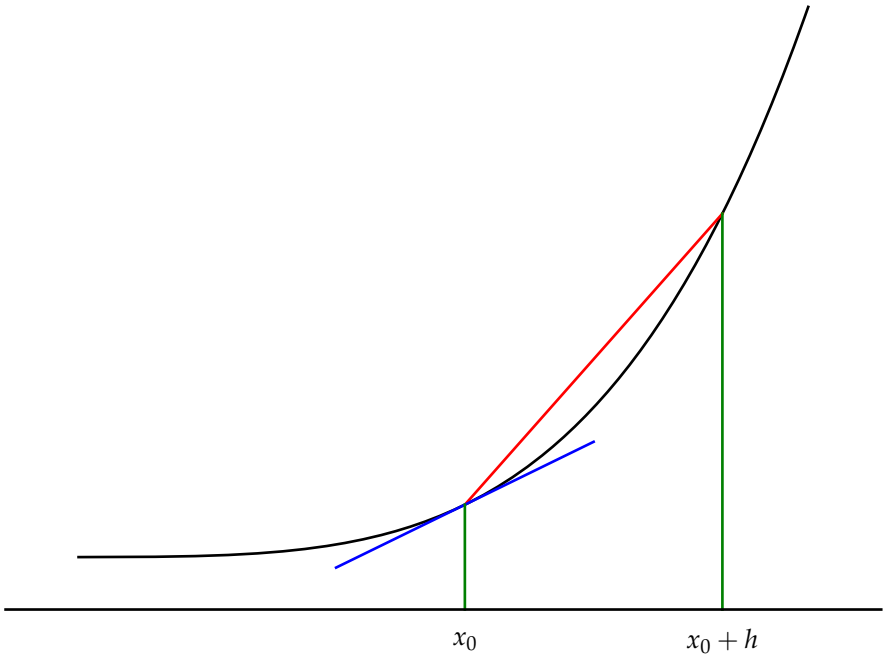

Figure 5.1: Forward difference method for approximating $f'(x_0)$. The derivative is approximated by the slope of the red line, while the true derivative is the slope of the blue line.

5.2 Central differencing

Compared with our algorithms for numerical integration, a first order method is not very impressive. In an attempt to derive a more accurate method, we can observe that all of the higher order integration methods have quadrature coefficients which are symmetric on each interval. Let's then try to form a differentiation rule by applying more symmetry. To do this, we'll try using two Taylor series expansions, one in the positive x direction from x_0 and one in the negative direction. Because we hope to achieve better than first order, we include an extra term in the series:

$$f(x_0 + h) = f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x_0) + \mathcal{O}(h^3) \tag{5.6}$$

$$f(x_0 - h) = f(x_0) - hf'(x_0) + \frac{(-h)^2}{2}f''(x_0) + \mathcal{O}((-h)^3) \tag{5.7}$$

Using the fact that $(-h)^2 = h^2$ and the absolute value signs from the definition of $\mathcal{O}$, this is equivalent to:

$$f(x_0 + h) = f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x_0) + \mathcal{O}(h^3) \tag{5.8}$$

$$f(x_0 - h) = f(x_0) - hf'(x_0) + \frac{h^2}{2}f''(x_0) + \mathcal{O}(h^3) \tag{5.9}$$

Remember that we are looking for an expression for $f'(x_0)$. Noticing the sign change between the derivative terms in the two equations, we subtract equation 5.9 from

equation 5.8 to give:

$$f(x_0 + h) - f(x_0 - h) = 2hf'(x_0) + \mathcal{O}(h^3) \tag{5.10}$$

Rearranging a little, we have:

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h} + \mathcal{O}(h^2) \tag{5.11}$$

This indicates that we have been successful: by taking an interval symmetric about x_0, we have created a second order approximation for the derivative of f. This symmetry gives the scheme its name: the central difference method. Figure 5.2 illustrates this scheme.

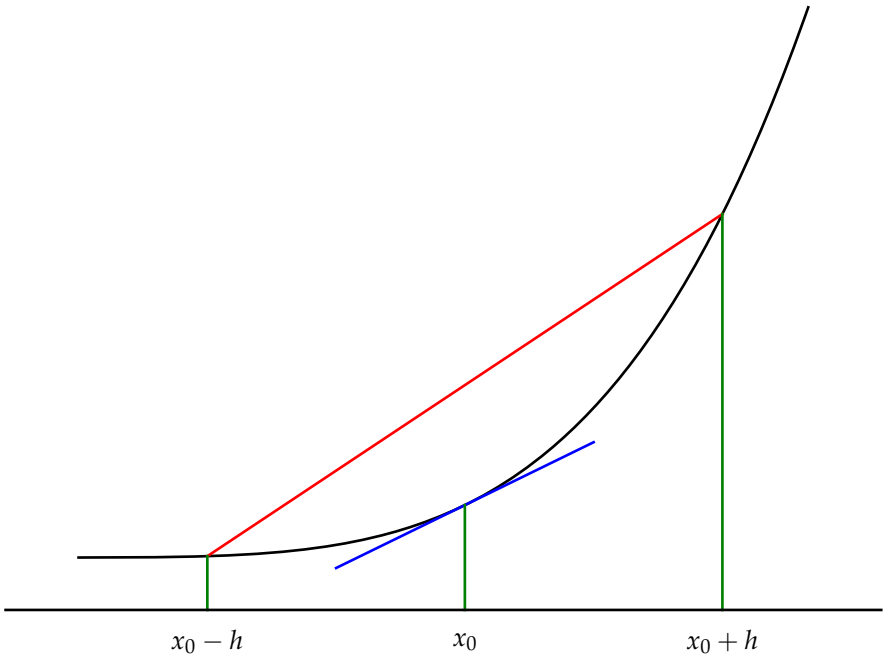

Figure 5.2: Central difference method for approximating $f'(x_0)$. The derivative is approximated by the slope of the red line, while the true derivative is the slope of the blue line.

Exercise 5.1. Create a python module named `differentiation` containing a function `forward_difference(f,x,h)` which uses the forward difference method to approximate the derivative of f at the point x using a step size of h.

Exercise 5.2. Add a second function `central_difference(f,x,h)` to your `differentiation` module. This should approximate the derivative of f at x using the central difference method with the value of h supplied.

You can use the script `test_derivative.py` from the `/numerical-methods-1` directory to test your differentiation functions.

Exercise 5.3. Write a Python script `plot_derivative_error.py` which uses your `differentiation` module to calculate the derivative of $f(x) = e^x$ at the point $x = 1$. Your script should plot the error in the derivative of f at $x = 1$ with 100 values of h between 0.0001 and 0.1 for each of the forward difference method and the central difference method.

Label the x axis h and the y axis *error* using the `pylab.xlabel` and `pylab.ylabel` commands. Also include a legend labelling the lines corresponding to central and forward difference methods using the `pylab.legend` command. Give your figure a suitable title using `pylab.title`.

Submit your `plot_derivative_error.py` script. There is no need to submit a printout of the plot.

5.3 Calculating second derivatives

Numerical differentiation may be extended to the second derivative by noting that the second derivative is the derivative of the first derivative. That is, if we define:

$$g(x) = f'(x) \tag{5.12}$$

then

$$f''(x) = g'(x) \tag{5.13}$$

We have noted above that the central difference method, being second order, is superior to the forward difference method so we will choose to extend that.

In order to calculate $f''(x_0)$ using a central difference method, we first calculate $f'(x)$ for each of two half intervals, one to the left of x_0 and one to the right:

$$f'(x_0 + \frac{h}{2}) \approx \frac{f(x_0 + h) - f(x_0)}{h} \tag{5.14}$$

$$f'(x_0 - \frac{h}{2}) \approx \frac{f(x_0) - f(x_0 - h)}{h} \tag{5.15}$$

We can now calculate the second derivative using these two values. Note that we know $f'(x)$ at the points $x_0 \pm \frac{h}{2}$, which are only h rather than $2h$ apart. Hence:

$$f''(x_0) \approx \frac{f'(x_0 + \frac{h}{2}) - f'(x_0 - \frac{h}{2})}{h} \tag{5.16}$$

$$\approx \frac{\frac{f(x_0+h)-f(x_0)}{h} - \frac{f(x_0)-f(x_0-h)}{h}}{h} \tag{5.17}$$

$$\approx \frac{f(x_0 + h) - 2f(x_0) + f(x_0 - h)}{h^2} \tag{5.18}$$

Exercise 5.4. By substituting power series for $f(x + h)$ and $f(x - h)$ into the right hand side of equation 5.18, prove that the central difference method for the second derivative is $\mathcal{O}(h^2)$.

5.4 A very brief primer on numerical methods for ODEs*

One of the most important applications of numerical mathematics in the sciences is the numerical solution of differential equations. This is a vast topic which rapidly becomes somewhat advanced, so we will restrict ourselves here to a very brief introduction to the solution of first order ordinary differential equations (ODEs). A much more comprehensive treatment of this subject is to be found in the Numerical Methods 2 module.

Suppose we have the general first order ODE:

$$x'(t) = f(x,t) \tag{5.19}$$
$$x(t_0) = x_0 \tag{5.20}$$

That is, the derivative of x with respect to t is some known function of x and t, and we also know the initial condition of x.

If we manage to solve this equation analytically, the solution will be a function $x(t)$ which is defined for every $t > t_0$. In common with all of the numerical methods we have encountered in this module, our objective will be to find an approximate solution to equation 5.19 at a finite set of points. In this case, we will attempt to find approximate solutions at $t = t_0, t_0 + h, t_0 + 2h, t_0 + 3h, \ldots$.

It is frequently useful to think of the independent variable, t, as representing time. A numerical method steps forward in time units of h, attempting to calculate $x(t+h)$ in using the previously calculated value $x(t)$.

5.4.1 Euler's method*

To derive a numerical method, we can first turn once again to the Taylor series. In this case, we could write:

$$x(t+h) = x(t) + hx'(t) + \mathcal{O}(h^2) \tag{5.21}$$

Using the definition of our ODE from equation 5.19, we can substitute in for $x'(t)$:

$$x(t+h) = x(t) + hf(t, x(t)) + \mathcal{O}(h^2) \tag{5.22}$$

Notice that the value of x used in the evaluation of f is that at time t. This simple scheme is named *Euler's method* after the 18th century Swiss mathematician, Leonard Euler.

The formula given is used to calculate the value of x one time step forward from the last known value. The error is therefore the local truncation error. If we actually wish to know the value at some fixed time T then we will have to calculate $(T - t_0)/h$ steps of the method. Using the same argument presented in section 4.6.2, this sum over $\mathcal{O}(1/h)$ steps results in a global truncation error for Euler's method of $\mathcal{O}(h)$. In other words, Euler's method is first order.

To illustrate Euler's method, and convey the fundamental idea of all time stepping methods, we'll use Euler's method to solve one of the simplest of all ODEs:

$$x'(t) = x \tag{5.23}$$
$$x(0) = 1 \tag{5.24}$$

We know, of course, that the solution to this equation is $x = e^t$, but let's ignore that for one moment and evaluate $x(0.1)$ using Euler's method with steps of 0.05. The first step is:

$$x(0.05) \approx x(0) + 0.05x'(0) \tag{5.25}$$
$$\approx 1 + .05 \times 1 \tag{5.26}$$
$$\approx 1.05 \tag{5.27}$$

Now that we know $x(0.05)$, we can calculate the second step:

$$x(0.1) \approx x(0.05) + 0.05x(0.05) \tag{5.28}$$
$$\approx 1.05 + .05 \times 1.05 \tag{5.29}$$
$$\approx 1.1025 \tag{5.30}$$

Now the actual value of $e^{0.1}$ is around 1.1051 so we're a couple of percent off even over a very short time interval.

Exercise 5.5. * Write a python module named `ode` containing a function `ode.euler(f,x_0,t_0,T,h)` which calculates the approximate solution of the system:

$$x'(t) = f(x,t)$$
$$x(t_0) = x_0$$

using steps of size h from t_0 to T. Your function should return `(t,x)` where `t` is the array $t, t+h, t+2h \ldots T$ and `x` is the corresponding array of approximate solutions.

You could test your function by solving the simple ODE above using decreasing values of h and checking that you converge to the analytic solution at $\mathcal{O}(h)$.

5.4.2 Heun's method*

Euler's method is first order because it calculates the derivative using only the information available at the beginning of the time step. As we observed for numerical integration and differentiation, higher order convergence can be obtained if we also employ information from other points in the interval. The method known as Heun's method may be derived by attempting to use derivative information at both the start and the end of the interval:

$$x(t+h) \approx x(t) + \frac{h}{2}\left(x'(t) + x'(t+h)\right) \tag{5.31}$$
$$\approx x(t) + \frac{h}{2}\big(f(x,t) + f(t+h, x(t+h))\big) \tag{5.32}$$

The difficulty with this approach is that we now require $x(t+h)$ in order to calculate the final term in the equation, and that's what we set out to calculate so we don't know it yet! The solution to this dilemma adopted in Heun's method is to use a first guess at $x(t+h)$ calculated using Euler's method:

$$\tilde{x}(t+h) = x(t) + hf(x,t) \tag{5.33}$$

$$x(t+h) \approx x(t) + \frac{h}{2}\big(f(x,t) + f(t+h, \tilde{x}(t+h))\big) \tag{5.34}$$

The generic term for schemes of this type is *predictor-corrector*. The initial calculation of $\tilde{x}(t+h)$ is used to predict the new value of x and then this is used in a more accurate calculation to produce a more correct value. We will state without derivation or proof that the global truncation error in Heun's method is $\mathcal{O}(h^2)$.

Exercise 5.6. * Add a function `heun(f,x_0,t_0,T,h)` to your `ode` module. The function should solve the same problem as `ode.euler` but use Heun's method rather than Euler's method.

Exercise 5.7. * Produce a Python script `plot_ode.py` which uses `ode.euler` and `ode.heun` to produce a plot comparing Euler's method and Heun's method to the analytic solution of the ODE:

$$x'(t) = x \tag{5.35}$$

$$x(0) = 1 \tag{5.36}$$

Use $h = 0.5$ and calculate the solution in the range $[0, 2.5]$. Ensure that your plot has labelled x and t axes, a title and a legend. If you choose to submit an answer to this exercise, you need only submit your Python code, it is not necessary to print the plot.

Chapter 6

Root finding

One of the most common mathematical tasks we encounter is the need to solve equations. That is to say, for some function $f(x)$ and a value b, we wish to know for which x it is true that $f(x) = b$.

In fact, this problem can be reduced to the special case of finding the values of x for which a given function takes the value zero. Suppose we wish to solve $f(x) = b$. We can simply define a new function $g(x) = f(x) - b$ with the result that our problem is now to solve $g(x) = 0$.

Quite a lot of the algebra which you learned at school is directed to solving this problem for particular sorts of functions. For example, if $g(x)$ is of the form $ax^2 + bx + c$ then the quadratic formula can be used to solve $g(x) = 0$. However for some functions an algebraic solution may be difficult to find or may not even exist, or we might not have an algebraic representation of our function at all! In these cases, how do we solve the equation?

6.1 The bisection method

The simplest root-finding algorithm is the bisection method. It relies on the fact that, for a continuous function f, if $f(x_0) < 0$ and $f(x_1) > 0$ then $f(x') = 0$ for some x' in the interval (x_1, x_2). This seemingly obvious result is known as the Bolzano theorem after the 18th-19th century Bohemian mathematician Bernhard Placidus Johann Nepomuk Bolzano. It is a special case of the intermediate value theorem.

The basic idea of the algorithm is that we know that the root lies in the interval $[x_l, x_r]$. We can now successively halve this interval and check in which half the root now lies. We keep the half of the interval in which the root is to be found and discard the other half. By repeating this operation, we obtain successively smaller intervals which contain the root. Figure 6.2 illustrates the application of the bisection method.

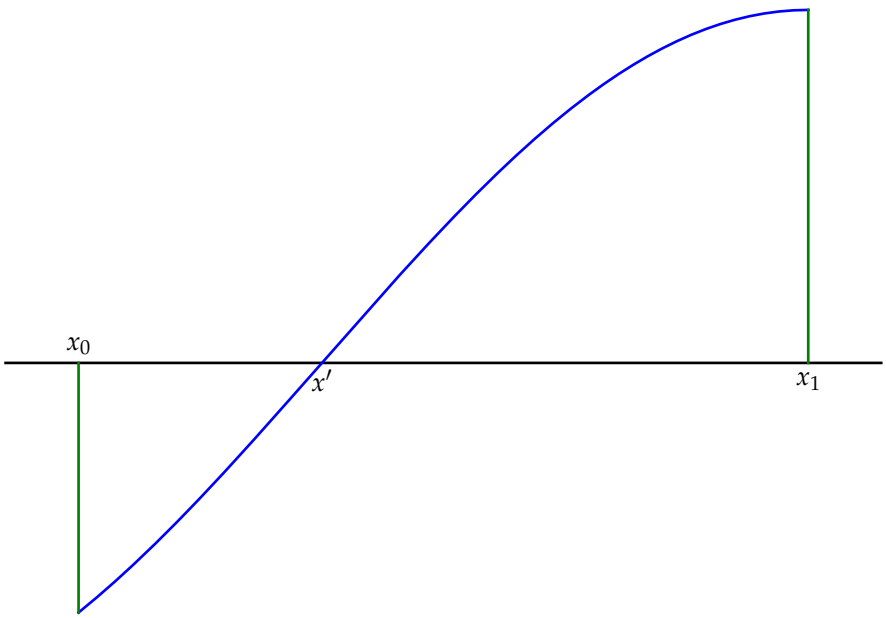

Figure 6.1: The Bolzano theorem: if $f(x_0) < 0$ and $f(x_1) > 0$ then there must exist a point x' such that $f(x') = 0$

6.1.1 Implementing the bisection method

We can write out the bisection algorithm using a notation known as *pseudocode*. Pseudocode is a mechanism for writing out sequences of mathematical operations without writing the particular syntactic details of a given programming language. Assume we have x_l and x_r such that $f(x_l)f(x_r) < 0$. That is to say, $f(x_l)$ and $f(x_r)$ have different signs. The bisection method is then:

$x_c \leftarrow \frac{x_l + x_r}{2}$
if $f(x_c)f(x_l) > 0$ **then**
  $x_l \leftarrow x_c$
else
  $x_r \leftarrow x_c$

The only symbol here which we're not already familiar with is $\leftarrow$ which is pseudocode for assigment. In Python, that's the same as $=$.

The algorithm above is just one step of the bisection method. How many steps should we take? The usual answer is that we will accept an approximate root x' if $|f(x')| < \epsilon$ for some small ϵ which we specify. We can therefore expand our algorithm to include multiple steps and a stopping criterion:

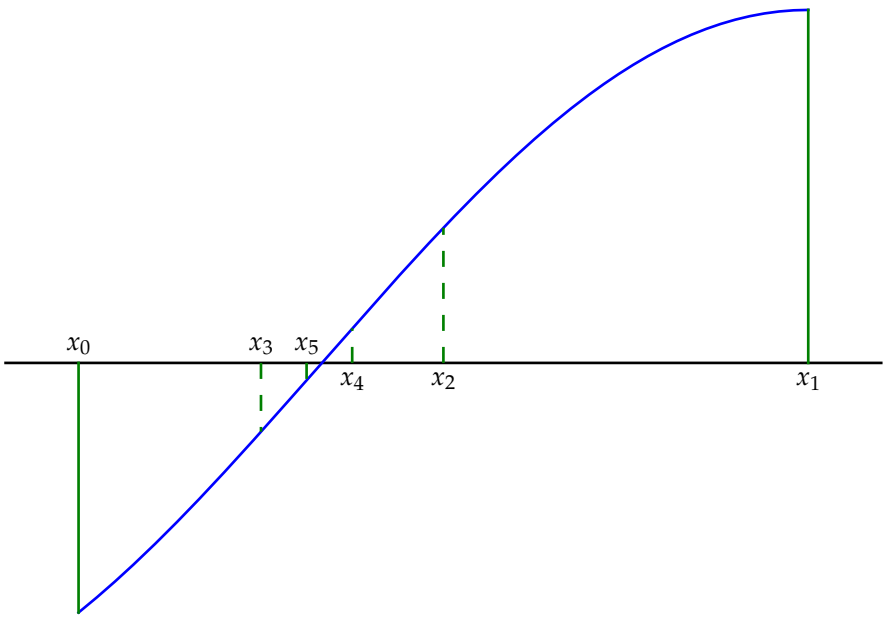

Figure 6.2: The bisection method finds the root by successively halving an interval known to contain that root. The process starts with the interval $[x_0, x_1]$; the subsequent x_n are the results of the method.

repeat
  $x_c \leftarrow \frac{x_l + x_r}{2}$
  if $f(x_c)f(x_l) > 0$ **then**
    $x_l \leftarrow x_c$
  else
    $x_r \leftarrow x_c$
until $|f(x_c)| < \epsilon$

Note that the pseudocode explains the steps to be taken by an algorithm but doesn't necessarily line up one for one with statements in a particular language. For example, the repeat... until construct needs to be translated to an appropriate Python loop.

The algorithm so far evaluates $f(x_l)$ on every step. This is fine if f is a simple function which is cheap to evaluate, but f might just as well be a huge and complex piece of code with a massive computation cost. We might therefore choose to introduce some temporary variables to prevent unnecessary re-evaluation of f:

$f_l \leftarrow f(x_l)$
repeat
  $x_c \leftarrow \frac{x_l + x_r}{2}$
  $f_c \leftarrow f(x_c)$
  if $f_c f_l > 0$ **then**
    $x_l \leftarrow x_c$
    $f_l \leftarrow f_c$
  else

$x_r \leftarrow x_c$
until $|f(x_c)| < \epsilon$

Exercise 6.1. Create a Python module `root_finding` containing a function `bisection(f,x_0,x_1,epsilon,record)`. This function should solve $f(x) = 0$ starting with the interval $[x_0, x_1]$ and applying the bisection method using `epsilon` as the convergence threshold ϵ. If `record` is `False`, then the function should return the final x_c: the best estimate of the root. If `record` is `True` then the function should return a sequence of pairs $(x_c, f(x_c))$ constructed from each successive estimate of the root. The default value of `record` should be `False`.

If if $f(x_0)$ and $f(x_1)$ have the same sign, your function should raise `ValueError` with an appropriate error message. If $\epsilon \leqslant 0$, your function should also raise `ValueError`, but with a message appropriate to that error.

You can test your function using `test_root_finding.py` from `/numerical-methods-1` by typing:

```
> python test_root_finding.py bisection
```

6.2 Iterative algorithms

The bisection method is an example of an *iterative algorithm*. That is to say, it is a process which produces a series of better and better approximations to the correct answer which converge to the exact answer as the sequence becomes infinitely long. We might question what use an algorithm is which might only produce the right answer if we run it for ever. However in reality, almost no quantities in science are known exactly: if there is already error in the input data and rounding error in the calculations then any answer we give will by necessity to be approximate. Iterative methods can be a very powerful tool in very quickly producing an approximate answer to a given accuracy.

6.3 Newton's method

Newton's method, also known as Newton-Raphson iteration is an iterative root-finding algorithm which uses the derivative of the function to provide more information about where to search for the root. We can derive Newton's method using the first two terms of the Taylor series for a function f starting at our initial guess x_1:

$$f(x_1 + h) = f(x_1) + hf'(x_1) + \mathcal{O}(h^2) \tag{6.1}$$

We wish to know what we need to set h to so that $f(x_1 + h) = 0$ so we impose this condition and solve for h:

$$h \approx -\frac{f(x_1)}{f'(x_1)} \tag{6.2}$$

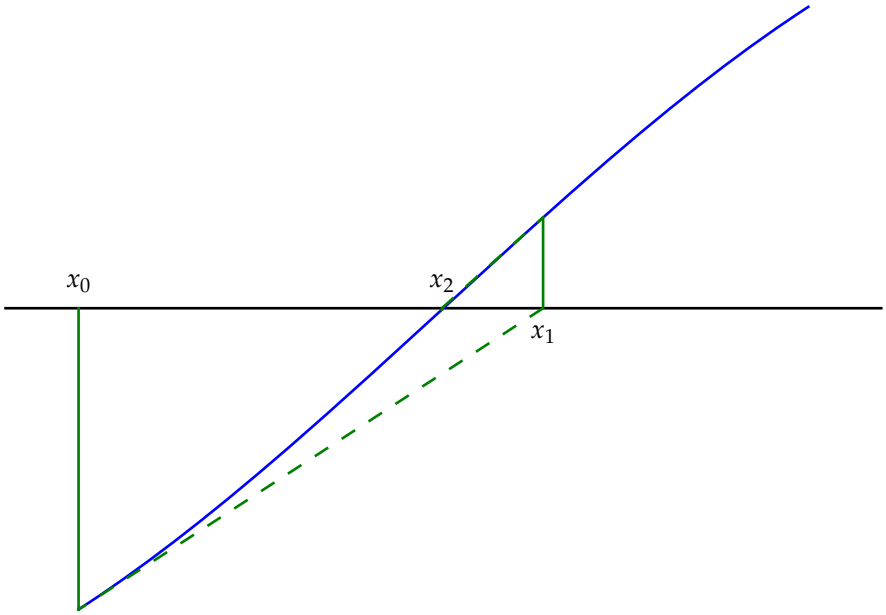

Figure 6.3: Newton's method finds the root by approximating $f(x)$ by a straight line at each iteration.

To form an iterative method, we use h to update the last estimate of the root:

$$x_{n+1} = x_n + h \tag{6.3}$$

$$= x_n - \frac{f(x_n)}{f'(x_n)} \tag{6.4}$$

Figure 6.3 illustrates the application of this method. It is noticable that x_2 is far closer to the actual root than even x_5 for the bisection method on the same problem.

6.3.1 Implementing Newton's method

Newton's method may be written in pseudocode as:

repeat
  $f_x \leftarrow f(x)$
  $x \leftarrow x - \frac{f_x}{f'(x)}$
until $(|f_x| < \epsilon)$ or maximum iterations exceeded.

Once again we have used a temporary variable f_x to avoid evaluating $f(x)$ too many times. Note that we have included a maximum iteration count since Newton's method is not guaranteed to converge.

Exercise 6.2. Add a function `newton(f, dfdx, x_0, epsilon, N, record)` to your `root_finding` module. The arguments have the same meaning as those for the `bisection`

function with the addition of `dfdx`, which should be the function which is the first derivative of `f`, and `N`, which is the maximum permitted number of iterations. The default value of `N` should be 100 and your function should raise `ArithmeticError` with an appropriate error message if this number of iterations is exceded. The output should depend on `record` in exactly the same manner as that of `bisection`.

Test your implementation using the `test_root_finding.py` script, passing `newton` as the argument instead of `bisection`

Exercise 6.3. Newton's method requires the derivative of the function to be known. Instead, we can use the last two iterations to approximate the derivative:

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

This leads to the secant method:

$$x_{n+1} = x_n - f(x_n)\left(\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}\right)$$

Write a function `secant(f, x0, x1, epsilon, N, record)` which uses the secant method to solve $f(x) = 0$. The function arguments should have the same meaning as in the `newton` function except that `x0` and `x1` are the two starting points and, of course, there is no need for `dfdx`.

Your secant function can be tested using the `test_root_finding.py` script with an argument of `secant`.

6.4 Rates of convergence

We now have three different root finding algorithms: the bisection method, Newton's method and the secant method from exercise 6.3. Which of these should we choose if we need to find a root? Part of the answer is that we might wish to choose the method which converges in the smallest number of iterations. In other words we want to study the rate of convergence.

The measure of convergence relevant to sequences of iterations is called Q-convergence because it is calculated from the quotient of adjacent sequence iterations.

Definition 6.1. A sequence x_k converges to x_* with Q-convergence order p if:

$$lim_{k\to\infty} \frac{|x_{k+1} - x_*|}{|x_k - x_*|^p} < 1 \tag{6.5}$$

For example, if a sequence converges Q-quadratically then for sufficiently large k, the error at least squares for each iteration. The Q-convergence order (Q-order) of a sequence is the largest p for which the sequence exhibits Q-convergence.

Note that the convergence rate is formally achieved in the infinite limit. What this is in essence saying is that the first iterations of the sequence may converge at other rates but they should settle down to the expected rate of convergence.

We are interested in numerically estimating the order of convergence. We assume therefore that there is a p and a μ such that:

$$\frac{|x_{k+1} - x_*|}{|x_k - x_*|^p} \approx \mu \tag{6.6}$$

In other words we assume that by setting k high enough we have approximately converged to the limit case. Rearranging, we have:

$$|x_{k+1} - x_*| \approx \mu |x_k - x_*|^p \tag{6.7}$$

We can also apply the same formula to the previous iteration in the sequence:

$$|x_k - x_*| \approx \mu |x_{k-1} - x_*|^p \tag{6.8}$$

By dividing these two approximate equations, we can eliminate μ:

$$\frac{|x_{k+1} - x_*|}{|x_k - x_*|} \approx \left(\frac{|x_k - x_*|}{|x_{k-1} - x_*|} \right)^p \tag{6.9}$$

Recall that we're looking for an expression for p so we take the logarithm of each side:

$$\log \left(\frac{|x_{k+1} - x_*|}{|x_k - x_*|} \right) \approx p \log \left(\frac{|x_k - x_*|}{|x_{k-1} - x_*|} \right) \tag{6.10}$$

Hence:

$$p \approx \frac{\log \left(\frac{|x_{k+1} - x_*|}{|x_k - x_*|} \right)}{\log \left(\frac{|x_k - x_*|}{|x_{k-1} - x_*|} \right)} \tag{6.11}$$

This expression still assumes that we know precisely the root to which the sequence is converging. In practice, we usually only have the sequence of iterations we have generated. Provided our sequence is well converged, we might choose to use the last member of the sequence to approximate the "true" limit value. Using the Pythonic notation of -1 to indicate the last index in a sequence, this gives the formula:

$$p \approx \frac{\log \left(\frac{|x_{k+1} - x_{-1}|}{|x_k - x_{-1}|} \right)}{\log \left(\frac{|x_k - x_{-1}|}{|x_{k-1} - x_{-1}|} \right)} \tag{6.12}$$

Exercise 6.4. Add a function `q_convergence(s)` to your `convergence` module. `s` is sequence of values. The return value should be a list of p values calculated from the input according to equation (6.12).

The module `test_q_convergence` in `/numerical-methods-1` contains two sequences `S1` and `S2` which you can use to check your work. `S1` converges at a Q-order approaching 1 while `S2` converges at a Q-order approaching 2. You can automatically apply these tests by running `test_q_convergence.py` as a script.

Exercise 6.5. Write a script `find_sin_root.py` which uses each of your three root finding methods to solve the equation:

$$x^5 = 1$$

For the bisection method, use $[0, 1.5]$ as the starting interval. Start Newton's method from $x_0 = 0$ and use $x_1 = 0.25$ for the secant method.

For each method, print the sequence of approximate roots generated. Also print estimates the order of convergence of each method on each function. You may need to make ϵ very small to get really good convergence.

6.5 Breakdown of Newton's method*

In exercise 6.5 you will have observed that Newton's method converges faster than the bisection method. This more rapid convergence comes, however, at a cost. While the bisection method is guaranteed to converge to the root and its accuracy is limited only by the accuracy with which the function f can be evaluated, there are a number of circumstances in which Newton's method can converge only slowly or even not at all.

6.5.1 Slow convergence*

At a multiple root, such as the root $x = 0$ for $f(x) = x^2$, the derivative of the function $f'(x)$ is zero. This makes the calculation of h less accurate as both the numerator and the denominator in equation (6.3) are tending to zero. The result of this is that Newton's method converges at first order instead of its usual second order.

Exercise 6.6. * Use Newton's method to solve $x^2 = 0$ with an initial condition of $x = 1$. Calculate the rate of convergence numerically.

6.5.2 Finding the wrong root*

If a function has more than one root, then an initial guess too far from the root may result in a very large first step which places the next iterate closer to another root. In this case, Newton's method may well converge to a root other than the one sought.

Exercise 6.7. * Use Newton's method to solve $\sin x = 0$ with a starting guess of $x = 1.7$. To which answer does the sequence converge?

6.5.3 Divergence*

If a function becomes very flat as its value becomes large, then Newton's method may set off in the wrong direction and diverge to infinity.

Exercise 6.8. * Use Newton's method to attempt to solve $\arctan x = 0$ starting at $x = 1.5$. The derivative of $\arctan x$ is $1./(1. + x^2)$.

6.6 Summary of root finding algorithms

We have met three root-finding algorithms here. They illustrate trade-offs which are common in numerical computing.

The bisection method requires minimal information about the function and always converges, but has a slow rate of convergence.

Newton's method often converges very fast but may converge slowly or not at all. It also requires us to know the gradient of the function we are solving.'

The secant method comes between the previous two methods in speed and does not require us to know the gradient of the function but can break down in the same ways as Newton's method.

Chapter 7

Numerical linear algebra

We met matrices in Maths Methods I and learned about matrix multiplication and how to solve linear systems by means of matrices. The linear systems encountered in science can be huge: many millions of rows and columns. It is therefore very important to have techniques for matrix operations on the computer.

7.1 Arrays: Python for vectors and matrices

In chapter 4 of Langtangen, which you have already covered in Programming for Geoscientists, you met Python `array` objects. These can be used to work with vectors and matrices. To do this we need to introduce the concept of the *rank* of an array. To put it simply, the rank is the number of indices which you need to describe the position of the entries in an array. A vector is an array with rank one while a matrix is an array with rank two[1].

This brings us to one of the most confusing terms in the array nomenclature: *dimension*. To a physicist, this is usually the length of a vector. However in computing it is usually used to refer to which of the indices of an array we are referring to. So a matrix (that is, a rank 2 array) has two dimensions and a vector (of any length) just the one.

The physicist's dimension is known as *extent* in computing. An array has an extent in each (computing) dimension. At this stage, an example might help:

```
In [133]: from numpy import *

In [134]: A=array([[1., 2.],[3., 4.],[5., 6.]])

In [135]: print(A)
[[ 1.  2.]
[ 3.  4.]
```

[1] Array is a computing term. Mathematicians tend to use the word *tensor*, which you might encounter in a later module

```
[ 5.  6.]]

In [136]: rank(a)
Out[136]: 2
```

`A` is a rank 2 array (i.e. a matrix). Its extent in dimension 0 is 3 and its extent in dimension 1 is 2. The vector composed of the extent of an array in each dimension is called its *shape*. To return to our example:

```
In [157]: A.shape
Out[157]: (3, 2)
```

7.1.1 Array notation and counting from zero

It is a widely used convention in mathematics that matrices and vectors are typeset with particular character types and weights. This makes formulae easier to read as it is possible to see the type of a symbol from its typography. We will use upright capital letters for matrices (for example A). Vectors will be given by bold lower case letters (**a**). There are two common uses of subscripts, and this can lead to some confusion. A bold lower case letter with a subscript, such as $\mathbf{a}_i$ indicates the i-th member of of a set of vectors. We might write the whole set as $\{\mathbf{a}_i\}$. A subscript can also be used to indicate the i-th component of a vector. This is, naturally, a scalar so we write this in italic: a_i is the i-th component of the vector **a**.

Python arrays (as well as lists, tuples and everything else in Python) are enumerated from 0. This means that the first entry in a vector `x` is `x[0]`. This is a cause of potential confusion as mathematicians often number vectors and matrices from 1. To try to avoid this confusion, in this course we will generally number from zero both in mathematical notation (x_0) and Python syntax (`x[0]`)

7.1.2 Slicing and dicing

List slices were introduced in Langtangen section 2.1.9. They were extended to arrays in section 4.2.2 and to multidimensional arrays in section 4.6.

As a brief reminder, the colon is used to indicate a range of values. Suppose we have a matrix `A`. The first column of `A` is given by `A[:,0]` while the last row of of `A` is `A[-1,:]`.

Slices can also extend over part of a dimension. The slice can be thought of as starting *before* the number given. So if `x` is a vector then `x[0:]` is the whole vector but `x[:0]` is a zero length vector containing nothing at all!

7.1.3 Creating arrays

There are many ways to create new array objects in Python but two are particularly important. One, which we met above, is to cast a sequence to an array:

```
array([1.,2.,3.])
```

A rank 2 array can be created by casting a sequence of sequences:

```
array([[1.,2.],[3.,4.]])
```

In this case, the outermost sequence corresponds to the first dimension of the array. This means that the inner sequences correspond to the rows of a matrix, which is probably what you expect.

The second important mechanism is using the `numpy.zeros` function. This function returns an array with value 0.0 in every element and with shape given by the argument. For example, to create a 3-vector call:

```
zeros(3)
```

or to create a matrix with 3 rows and 4 columns, call:

```
zeros((3,4))
```

Note the second set of brackets: `zeros` takes exactly one argument but that argument can be a scalar or a (rank 1) sequence.

Once an array of zeros has been created, the entries can be populated by assigning singly or to slices.

7.2 Dot products and matrix multiplication

Let's think about the dot product and matrix multiplication operators which we met in maths methods 1. Assume that A and B are matrices and $\mathbf{x}$ and $\mathbf{y}$ are vectors and for simplicity lets assume that the matrices are square and the extent of both the matrices and the vectors is n. Then we can write down the following mathematical identities:

$$\mathrm{A}\mathbf{y} = \sum_{i=0}^{n-1} \mathrm{A}_{:,i} y_i \tag{7.1}$$

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=0}^{n-1} x_i y_i \tag{7.2}$$

$$\mathbf{x}\mathrm{A} = \sum_{i=0}^{n-1} x_i \mathrm{A}_{i,:} \tag{7.3}$$

$$\mathrm{AB} = \sum_{i=0}^{n-1} \mathrm{A}_{:,i} \mathrm{B}_{i,:} \tag{7.4}$$

The similarity between these statements is striking. In fact, the dot product and matrix multiplication are really the same operator. If `C` and `D` are arrays then we multiply them by writing `numpy.dot(C,D)`. The answer is achieved by summing over the last dimension of `C` and the first dimension of `D`. The operation is only defined if the extents of `C` and `D` match along these dimensions and an exception will be raised if they do not.

From this definition we can draw some other conclusions. The first is that:

```
rank(dot(C,D)) == rank(C) + rank(D) - 2
```

What does this mean if `C` and `D` are both vectors? Well vectors have rank 1 so this statement says that the rank of the dot product of two vectors is zero. This is correct. You can think of a scalar value as being a rank zero array. Python even agrees:

```
In [163]: rank(1.0)
Out[163]: 0
```

7.3 Linear systems

One of the most important classes of problems to be able to solve is the matrix equation:

$$\mathrm{A}\mathbf{x} = \mathbf{b} \tag{7.5}$$

in which A is known square matrix, **b** is a known vector and **x** is an unknown vector. Before we move on to methods for solving this sort of system, we'll spend some time on what this equation means and trying to understand the question it poses.

7.4 Span and linear independence

Let's look at the matrix A in a slightly different way. Rather than thinking of A as a set of row vectors, let's think of a as a collection of columns:

$$\mathrm{A} = \left[\begin{array}{c|c|c|c} & & & \\ \mathrm{A}_{:,0} & \mathrm{A}_{:,1} & \cdots & \mathrm{A}_{:,n} \\ & & & \end{array}\right] \tag{7.6}$$

the matrix product A**x** can then be thought of as a weighted sum of the columns of A:

$$\mathrm{A}\mathbf{x} = x_0\mathrm{A}_{:,0} + x_1\mathrm{A}_{:,1} + \cdots + x_n\mathrm{A}_{:,1} \tag{7.7}$$

This is also described as a *linear combination* of the columns of A.

What form to linear combinations of vectors take? If we have a single vector **a** then the linear combinations of that vector ($x\mathbf{a}$) are simply the line passing through that vector. Figure 7.1 illustrates this.

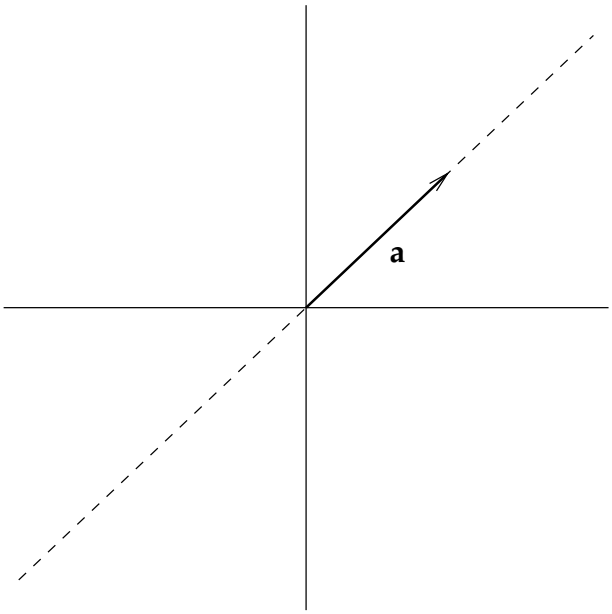

Figure 7.1: The span of a single vector, $\mathbf{a}$, is the set $x\mathbf{a}$ for all $x \in \mathbb{R}$.

If we have a set of vectors $\{\mathbf{a}_i\}$ then the set of linear combinations of those vectors is called the *span* of that set. The span of a single vector (other than the zero vector) is a line passing through that vector.

What about two vectors $\mathbf{a}_0$ and $\mathbf{a}_1$? Well now we have two options. Either $\mathbf{a}_0 = x\mathbf{a}_1$ for some $x \in \mathbb{R}$, or there is no such x. In the first case, the two vectors are colinear and their span is just the line in which they both lie. In the second case, the two vectors are not colinear and their span is the plane in which the two vectors lie. If the vectors are two-dimensional ($\mathbf{a}_0, \mathbf{a}_1 \in \mathbb{R}^2$) then that plane is simply equal to $\mathbb{R}^2$. On the other hand if the vectors lie in $\mathbb{R}^3$ or a higher dimensional space, then their span is the plane in that space passing through those two vectors.

For a set of three vectors, there are now three possibilities. The first is that all three vectors are scalar multiples of each other:

$$\mathbf{a}_0 = x_1\mathbf{a}_1 = x_2\mathbf{a}_2 \tag{7.8}$$

for some $x_1, x_2 \in \mathbb{R}$. In this case the span of the set is one-dimensional. The next possibility is that the vectors are not all colinear one of the vectors in the set can be written as a weighted sum of the other two:

$$\mathbf{a}_0 = x_1\mathbf{a}_1 + x_2\mathbf{a}_2 \tag{7.9}$$

for some $x_1, x_2 \in \mathbb{R}$. In this case the span is once again a plane and all three vectors lie in that plane. The final option is that none of the vectors can be written as a linear combination of the other two, in which case the span is three dimensional: either all of $\mathbb{R}^3$ if the vectors lie in $\mathbb{R}^3$ or a three-dimensional subspace of the space if larger.

If none of a set of vectors can be written as a linear combination of the others, then that set is said to be *linearly independent*.

Definition 7.1. A set of vectors $\{\mathbf{a}_0, \ldots, \mathbf{a}_n\}$ is *linearly dependent* if and only if there exists a set of coefficents $\{x_0, \ldots, x_n\}$ such that:

$$\sum_{i=0}^{n} x_i \mathbf{a}_i = 0 \tag{7.10}$$

and at least one of the coefficients $\mathbf{x}_i$ is not zero.

Definition 7.2. A set of vectors is *linearly independent* if and only if it is not *linearly dependent*.

Let's look back at our matrix A. We can rephrase the matrix equation:

$$\mathrm{A}\mathbf{x} = \mathbf{b} \tag{7.11}$$

as the question: which linear combination of the columns of A gives $\mathbf{b}$ As we've just seen, and as you learned in Mathematical Methods 1, the answer may be that *no* linear combination of the columns of A gives $\mathbf{b}$ or it may be that there is more than one combination which works.

If A is $n \times n$ then we've seen above that the span of its columns, which we write $\mathrm{span}(\mathrm{A})$ is a subspace of $\mathbb{R}^n$ with dimension somewhere between 0 and n. Further, the $\mathrm{span}(\mathrm{A})$ is only n-dimensional if the columns of A form a linearly independent set. This leads us to a whole list of properties which are equivalent for a $n \times n$ square matrix A:

1. The columns of A are linearly independent.
2. The span of the columns of A is $\mathbb{R}^n$.
3. A is invertible. i.e. there exists a matrix A^{-1} such that $\mathrm{A}\mathrm{A}^{-1} = \mathrm{I}$
4. $\mathrm{A}\mathbf{x} = \mathbf{b}$ has a unique solution for every $\mathbf{b} \in \mathbb{R}^n$.
5. The unique solution to $\mathrm{A}\mathbf{x} = \mathbf{0}$ is the zero vector, $\mathbf{0}$.

Using the knowledge of eigenvalues and determinants gained in Maths Methods I, we can add two more items to this list:

6. $\det(\mathrm{A}) \neq 0$.
7. 0 is not an eigenvalue of A.

As a reminder:

Definition 7.3. λ is an *eigenvalue* of a matrix A if there exists some non-zero vector $\mathbf{v}$ such that:

$$\mathrm{A}\mathbf{v} = \lambda\mathbf{v} \tag{7.12}$$

Exercise 7.1. If A is a square matrix whose columns are *not* linearly independent, show that the equation:

$$\mathrm{A}\mathbf{x} = \mathbf{0} \tag{7.13}$$

has a non-zero solution.

Exercise 7.2. Use the result from the previous exercise to argue that 0 is an eigenvalue of A.

7.4.1 Matrix rank

Unfortunately at this stage we encounter a very confusing clash of terminology. The dimension of the span of a matrix is known as the *rank* of that matrix. This has nothing to do with the concept of the rank of an array. All matrices are rank 2 arrays. We will therefore avoid talking about the matrix rank (i.e. the dimension of the column span) as much as possible. However, there is one important piece of terminology which it is hard to avoid. A matrix whose column span has the maximum possible dimension is said to have full rank. For a $n \times m$ matrix, this maximum rank is $\min(n, m)$.

We can now add an additional equivalent statement to the list above:

8. A has full rank.

7.5 Matrices as a basis for $\mathbb{R}^n$

Yet another way that we can think of the columns of a matrix A is as the axes of a coordinate system. In this interpretation, the matrix equation

$$\mathrm{A}\mathbf{x} = \mathbf{b} \tag{7.14}$$

means: what are the coordinates of the vector $\mathbf{b}$ in the coordinate system given by the columns of A?

To illustrate this, think of the matrix whose columns are the usual three-dimensional basis vectors $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$:

$$\left[\begin{array}{c|c|c} \mathbf{i} & \mathbf{j} & \mathbf{k} \end{array}\right] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \equiv \mathrm{I} \tag{7.15}$$

If we pose the equation:

$$\mathrm{I}\mathbf{x} = \mathbf{b} \tag{7.16}$$

then we are asking: what are the coordinates of $\mathbf{b}$ with respect to the usual basis? The answer is, of course, $\mathbf{x} = \mathbf{b}$.

So we can think of an $n \times n$ square matrix of full rank as an alternative set of basis vectors for $\mathbb{R}^n$.

7.6 Solving triangular systems

A *lower triangular matrix* is one in which all of the entries above the diagonal are zero.

$$\mathrm{L} = \begin{bmatrix} \mathrm{L}_{0,0} & 0 & \cdots & 0 \\ \mathrm{L}_{1,0} & \mathrm{L}_{1,1} & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ \mathrm{L}_{n,0} & \mathrm{L}_{n,1} & \cdots & \mathrm{L}_{n,n} \end{bmatrix} \tag{7.17}$$

Solving a matrix equation involving a lower triangular matrix is easy. Let's write:

$$\mathrm{L}\mathbf{x} = \mathbf{b} \tag{7.18}$$

We just have to remember the rule for matrix-vector multiplication. We first multiply the first row of L by $\mathbf{x}$ and set it equal to the first entry in $\mathbf{b}$:

$$\mathrm{L}_{0,0}x_0 + \mathbf{0} \cdot \mathbf{x}_{1:} = b_0 \tag{7.19}$$

Here we're writing $\mathbf{0}$ to mean a vector of all zeros and copying Python's array slice notation so $\mathbf{x}_{1:}$ means the second and subsequent entries in $\mathbf{x}$.

Hence:

$$x_0 = b_0 / \mathrm{L}_{0,0} \tag{7.20}$$

In the second row, we get:

$$\mathrm{L}_{1,0}x_0 + \mathrm{L}_{1,1}x_1 + \mathbf{0} \cdot \mathbf{x}_{3:} = b_1 \tag{7.21}$$

Since we already know x_0, we can write this as:

$$x_1 = \left(b_1 - \mathrm{L}_{1,0}x_0\right) / \mathrm{L}_{1,1} \tag{7.22}$$

This pattern continues: for each row we can use the fact that we already know all the preceding $\mathbf{x}$ values. The expression for the n-th row of L is therefore:

$$x_n = \left(b_n - \mathrm{L}_{n,:n} \cdot \mathbf{x}_{:n}\right) / \mathrm{L}_{n,n} \tag{7.23}$$

Exercise 7.3. Create a module called `linear_algebra`. Write a function `solve_lower(L,b)` which solves the matrix problem $\mathrm{L}\mathbf{x} = \mathbf{b}$ if L is lower triangular and returns the answer $\mathbf{x}$. If L is not lower triangular, you should raise a `ValueError` exception. *Hint:* you can check that you've got the right answer by calculating `dot(L,x)-b`.

Exercise 7.4. The functions in `/numerical-methods-1/random_matrices.py` are useful for generating matrices and vectors to test linear algebra problems. By using the test:

```
if __name__=='__main__':
    # Test code goes here.
```

it is possible to include code in your `linear_algebra.py` file which is only executed when the file is used as a script rather than as a module. Add test code in this form to `linear_algebra.py` which solves $L\mathbf{x} = \mathbf{b}$ for random lower triangular matrices L of various sizes and corresponding random vectors $\mathbf{b}$. You should check that the answer is correct and print appropriate messages for success or failure.

Exercise 7.5. An *upper triangular matrix* is, naturally, a matrix in which the entries below the diagonal are all zero. Write another function for your module, this time `solve_upper(U,b)` to solve an upper triangular problem. You should raise `ValueError` if the matrix U is not actually upper triangular. *Hint:* work backwards from the last row of the matrix.

Extend your test code to also test `solve_upper`.

7.7 The LU factorisation

We now know how to solve problems of the form $L\mathbf{x} = \mathbf{b}$ and $U\mathbf{x} = \mathbf{b}$ where L and U are lower and upper triangular respectively. This might not seem like a very useful piece of information since most matrices we are likely to encounter in practice are unlikely to be lower or upper triangular. Suppose, though, that we were able to write a general matrix A in the form:

$$A = LU \tag{7.24}$$

Then we could rewrite the problem

$$A\mathbf{x} = \mathbf{b} \tag{7.25}$$

as:

$$LU\mathbf{x} = \mathbf{b} \tag{7.26}$$

Now if we create a new unknown vector $\mathbf{c}$ which we define by:

$$U\mathbf{x} = \mathbf{c} \tag{7.27}$$

then we can substitute into equation (7.26) to create:

$$L\mathbf{c} = \mathbf{b} \tag{7.28}$$

We know how to solve equation (7.28) for $\mathbf{c}$. Once we have $\mathbf{c}$ then we know how to solve equation (7.27) for $\mathbf{x}$.

The challenge, then is to find a way to rewrite A as LU.

7.7.1 Gaussian elimination revisited

In Maths Methods 1 you learned how to solve the matrix equation $Ax = b$ using Gaussian elimination. We'll revisit this algorithm now from the perspective of matrix operations as a way of forming our LU factorisation.

Let's take the following matrix as our example:

$$A = \begin{bmatrix} 5 & 7 & 5 & 9 \\ 5 & 14 & 7 & 10 \\ 20 & 77 & 41 & 48 \\ 25 & 91 & 55 & 67 \end{bmatrix} \tag{7.29}$$

Remember that the first step of Gaussian elimination is to set the sub-diagonal rows in the first column to zero by subtracting multiples of the first row from each of the subsequent rows. We can write this as a matrix multiplication:

$$\mathrm{L}_0 \cdot \mathrm{A} \tag{7.30}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ -4 & 0 & 1 & 0 \\ -5 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 5 & 7 & 5 & 9 \\ 5 & 14 & 7 & 10 \\ 20 & 77 & 41 & 48 \\ 25 & 91 & 55 & 67 \end{bmatrix} = \begin{bmatrix} 5 & 7 & 5 & 9 \\ 0 & 7 & 2 & 1 \\ 0 & 49 & 21 & 12 \\ 0 & 56 & 30 & 22 \end{bmatrix}$$

To see how this works, let's remember the algorithm for matrix multiplication:

$$\mathrm{L_0A} = \sum_{i=1}^{n} \mathrm{L}_0[:, i]\mathrm{A}[i, :]. \tag{7.31}$$

If we restrict this to just the second row of the right hand side (row number 1), this becomes:

$$\sum_{i=1}^{n} \mathrm{L}_0[1, i]\mathrm{A}[i, :] = -1 \times \mathrm{A}[0, :] + 1 \times \mathrm{A}[1, :] \tag{7.32}$$

In other words, the second line of L_0 has the effect summing -1 times the first row of A and 1 times the second row. The resulting vector is the second row of the product $\mathrm{L}_0 \cdot \mathrm{A}$.

Similarly, the operation of the third row of L_0 is to form the third row of $\mathrm{L}_0 \cdot \mathrm{A}$ by adding -4 times the first row of A to 1 times the third row of A.

So what's the rule for generating L_0? We start with the identity matrix, because we always have one times the current row. We then need to subtract the correct multiple of the first row so that the first entry of the second and subsequent row of $\mathrm{L}_0 \cdot \mathrm{A}$ is zero. An algorithm for this is:

$\mathrm{L}_0 \leftarrow \mathrm{I}$
for $i \leftarrow 1$ **to** $\mathrm{len}(\mathrm{A}) - 1$ **do**
  $\mathrm{L}_0[i, 0] \leftarrow -\mathrm{A}[i, 0]/\mathrm{A}[0, 0]$

Now let's procede to the second row of A. This time we use Gaussian elimination to produce zeros below the diagonal in the *second* column:

$$\mathrm{L}_1 \cdot (\mathrm{L}_0 \cdot \mathrm{A}) \tag{7.33}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -7 & 1 & 0 \\ 0 & -8 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 5 & 7 & 5 & 9 \\ 0 & 7 & 2 & 1 \\ 0 & 49 & 21 & 12 \\ 0 & 56 & 30 & 22 \end{bmatrix} = \begin{bmatrix} 5 & 7 & 5 & 9 \\ 0 & 7 & 2 & 1 \\ 0 & 0 & 7 & 5 \\ 0 & 0 & 14 & 14 \end{bmatrix}$$

This time, Gaussian elimination was applied not to A but to the answer from the first step: $L_0 \cdot A$. So to write an algorithm for Gaussian elimination, we need to keep track of the previous steps applied to A. For reasons which will become obvious, let's write $U = L_0 \cdot A$. We could calculate U directly using a dot product, but that would involve a lot of useless multiplications by zero since L_0 is mostly made up of zeros. Instead, we can modify our algorithm for finding L_0 to also calculate U:

$L_0 \leftarrow I$
$U \leftarrow A$
for $i \leftarrow 1$ **to** $\text{len}(A) - 1$ **do**
  $L_0[i, 0] \leftarrow -U[i, 0]/U[0, 0]$
  $U[i, :] \leftarrow U[i, :] + L_0[i, 0] \times U[0, :]$

Now we can use that U and the corresponding algorithm to calculate L_1:

$L_1 = I$
for $i \leftarrow 2$ **to** $\text{len}(A) - 1$ **do**
  $L_1[i, 1] \leftarrow -U[i, 1]/U[1, 1]$
  $U[i, :] \leftarrow U[i, :] + L_0[i, 1] \times U[1, :]$

Notice that the loop over i now starts from 2 because we're setting the subdiagonal entries of the second row.

Of course we don't really want to write out a special case of the algorithm for each row of the matrix, and we can easily generalise this to the full algorithm for all the L_j:

$U \leftarrow A$
for $j \leftarrow 0$ **to** $\text{len}(A) - 2$ **do**
  $L_j \leftarrow I$
  for $i \leftarrow j + 1$ **to** $\text{len}(A) - 1$ **do**
    $L_j[i, j] \leftarrow -U[i, j]/U[j, j]$
    $U[i, :] \leftarrow U[i, :] + L_j[i, j] \times U[j, :]$

We can apply this algorithm to get the final step:

$$L_2 \cdot (L_1 \cdot (L_0 \cdot A)) \tag{7.34}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -2 & 1 \end{bmatrix} \times \begin{bmatrix} 5 & 7 & 5 & 9 \\ 0 & 7 & 2 & 1 \\ 0 & 0 & 7 & 5 \\ 0 & 0 & 14 & 14 \end{bmatrix} = \begin{bmatrix} 5 & 7 & 5 & 9 \\ 0 & 7 & 2 & 1 \\ 0 & 0 & 7 & 5 \\ 0 & 0 & 0 & 4 \end{bmatrix}$$

We can now see why we called the product matrix U. So we've now managed to

write:

$$L_2 \cdot (L_1 \cdot (L_0 \cdot A)) = U \tag{7.35}$$

This is sort of like our goal of $A = L \cdot U$ but not quite. To get from one to the other we first need to move the L_i from the left to the right of the equation. We can move L_2 by multiplying both sides by its inverse:

$$L_2^{-1} \cdot L_2 \cdot (L_1 \cdot (L_0 \cdot A)) = L_2^{-1} \cdot U \tag{7.36}$$

$$I \cdot (L_1 \cdot (L_0 \cdot A)) = L_2^{-1} \cdot U \tag{7.37}$$

$$(L_1 \cdot (L_0 \cdot A)) = L_2^{-1} \cdot U \tag{7.38}$$

By successively multiplying by the inverses if L_i in reverse order, we come to:

$$A = L_0^{-1} \cdot L_1^{-1} \cdot L_2^{-1} \cdot U \tag{7.39}$$

This looks a bit closer, but it depends on us knowing L_i^{-1}. This is the first piece of magic in this algorithm. Let's take the example of L_0. It so turns out that:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ -4 & 0 & 1 & 0 \\ -5 & 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 4 & 0 & 1 & 0 \\ 5 & 0 & 0 & 1 \end{bmatrix}. \tag{7.40}$$

To see why this is true, consider the $2, 0$ entry in $L_0 \cdot L_0^{-1}$:

$$L_0[2,0] = \begin{bmatrix} -4 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 4 \\ 5 \end{bmatrix} \tag{7.41}$$

$$= -4 \times 1 + 0 \times 1 + 1 \times 4 + 0 \times 5 \tag{7.42}$$

$$= 0 \tag{7.43}$$

This pattern of cancellations repeats throughout. We therefore have a nice easy way of forming L_i^{-1}.

The final stage of our quest is to combine $L_0^{-1} \cdot L_1^{-1} \cdot L_2^{-1}$ into a single matrix L, which we hope will be lower diagonal. At this point, a second piece of magic comes to our aid. It turns out that:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 4 & 0 & 1 & 0 \\ 5 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 7 & 1 & 0 \\ 0 & 8 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 4 & 7 & 1 & 0 \\ 5 & 8 & 2 & 1 \end{bmatrix} \tag{7.44}$$

That is, the sub-diagonal entries are merged. Note that this is not a general matrix property, it only applies to matrices of the form L_i and only when multiplied in this order.

We can use these inverse and multiplication properties to modify our algorithm. Notice that the sign changes on the entries of L and this causes a sign change on the construction of U to ensure we get the same U as before:

$U \leftarrow A$
$L \leftarrow I$
for $j \leftarrow 0$ **to** $\text{len}(A) - 2$ **do**
  for $i \leftarrow i + 1$ **to** $\text{len}(A) - 1$ **do**
    $L[i, j] \leftarrow U[i, j] / U[j, j]$
    $U[i, :] \leftarrow U[i, :] - L[i, j] \times U[j, :]$

Exercise 7.6. Add a function `factorise_lu(A)` to your `linear_algebra` module. This function should take a matrix `A` and return its LU factors `L` and `U`. Your function should raise `ValueError` if `A` is not square.

Extend the test code in `linear_algebra.py` to test `factorise_lu` for matrices of various sizes. In particular, `random_matrices.random_lu` is guaranteed to return a matrix for which LU factorisation will work.

Exercise 7.7. Add a function `solve_lu(A,b)` to your `linear_algebra` module. This function should take a matrix `A` and a vector `b` and return `x`, the solution to $A\mathbf{x} = \mathbf{b}$. Calculate the solution by first calling `factorise_lu` and then solving the two resulting diagonal systems.

Extend the test code in `linear_algebra.py` to test `solve_lu` for matrices of various sizes.

7.8 Breakdown of the LU factorisation

Let's take the example[2] of the matrix:

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \tag{7.45}$$

If we apply LU factorisation to this matrix then $U[0, 0] = 0$ and as soon as we start calculating the second row, we have to divide by $U[0, 0]$ so the algorithm fails.

There is an even nastier version of this problem. Consider:

$$B = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 1 \end{bmatrix} \tag{7.46}$$

If we apply LU factorisation to this matrix using exact arithmetic, we have:

$$L = \begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix}, \qquad U = \begin{bmatrix} 10^{-20} & 1 \\ 0 & 1 - 10^{20} \end{bmatrix} \tag{7.47}$$

The factorisation has numbers with huge and tiny magnitudes in it. This might not be a problem in exact arithmetic, but in floating point arithmetic the last entry in U

[2] This example is from Trefethen and Bau (1997)

is likely to be rounded to -10^{20}. Is this OK? After all 1 in 10^{20} error is pretty tiny. However:

$$\begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix} \times \begin{bmatrix} 10^{-20} & 1 \\ 0 & -10^{20} \end{bmatrix} = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 0 \end{bmatrix} \tag{7.48}$$

Look at the last entry in the product. That's a huge error.

The lesson we draw from this is that LU factorisation is *unstable*. It is succeptable to dividing by zero or by tiny numbers when these occur in the algorithm and the effects of very small rounding errors result in huge errors in the factorisation.

7.9 Pivoting*

If we were to factor matrix A from equation (7.45) using Gaussian elimination, we would immediately know what to do: switch the first two rows and the division by zero vanishes. How do we introduce row interchange operations in an automatic way?

7.9.1 Permutation matrices*

It turns out that we can swap the rows or columns of a matrix by multiplying by a special matrix. The mathematical term for switching the order of a list is *permutation*. A permutation matrix is one in which every row and every column contains exactly one entry with the value 1 and all the other entries zero. If we left multiply by a permutation matrix (i.e. evaluate PA then the matrix will rearrange the *rows* of A. In particular, if $\mathrm{P}_{i,j} = 1$ then the i-th row of PA will be equal to the j-th row of A. Some examples might help. The first is a 3×3 matrix which swaps the first two rows:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{7.49}$$

This next one is known as a cyclic permutation, it moves each row to the next one down and the last row to the first:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{7.50}$$

Note that the permutation matrix is just the identity matrix with its rows permuted. The identity matrix is itself a permutation: the identity permutation which maps each row to itself.

Another way of writing down a permutation is as a vector. The i-th entry of the vector is the number to which i is mapped under the permutation. For this last example, the permutation vectors corresponding to the permutation matrices above are:

$$\begin{bmatrix} 1, 0, 2 \end{bmatrix} \tag{7.51}$$

and

$$\begin{bmatrix}3,0,1,2\end{bmatrix} \tag{7.52}$$

Python makes it trivial to generate permutation matrices from permutation vectors. First, create an identity matrix using the `numpy.eye` function and then select the rows in the order of the permutation by providing the permutation vector as a subscript. For example, we can produce the first permutation matrix above by writing:

```
from numpy import array,eye
p_vector=array([1,0,2])
p_array=eye(3)
p_array=p_array[p_vector]
```

Actually, since the result of `eye` is an array, we can write:

```
from numpy import array,eye
p_vector=array([1,0,2])
p_array=eye(3)[p_vector]
```

7.9.2 Partial pivoting*

We can now apply permutation matrices to swap the rows of A so we never end up dividing by a value which is close to zero. The way we do this is by replacing our equation:

$$\mathrm{A}\mathbf{x} = \mathbf{b} \tag{7.53}$$

with

$$\mathrm{PA}\mathbf{x} = \mathrm{P}\mathbf{b} \tag{7.54}$$

where P is a permutation matrix which interchanges the rows of A to avoid unfortunate divisions. So far, so good. We know how to construct P if we know which row to choose. Let's look back at our LU algorithm:

$\mathrm{U} \leftarrow \mathrm{A}$
$\mathrm{L} \leftarrow \mathrm{I}$
for $j \leftarrow 0$ **to** $\mathrm{len}(\mathrm{A}) - 2$ **do**
  for $i \leftarrow j + 1$ **to** $\mathrm{len}(\mathrm{A}) - 1$ **do**
    $\mathrm{L}[i,j] \leftarrow \mathrm{U}[i,j]/\mathrm{U}[j,j]$
    $\mathrm{U}[i,:] \leftarrow \mathrm{U}[i,:] - \mathrm{L}[i,j] \times \mathrm{U}[j,:]$

The only division, and therefore the problem term, is the division by $\mathrm{U}[j,j]$. If we swap rows, this means we can instead choose to divide by any value in the column under $\mathrm{U}[j,j]$. In other words, $\mathrm{U}[k,j]$ for any $k \geqslant j$.

There is a catch, however. U changes at each step of the algorithm so it's only at the start of the step that we can decide which row to use next. To minimise the rounding errors from the division, we always choose the row for which $|\mathrm{U}[k,j]|$ is largest.

But wait, we've already started! We can't just go messing with the row order in U because we've already started building L. The answer is that it's all fine if we also

switch the rows in the part of L we've already built. This modifies L as if we had always assembled it in permuted order. The new algorithm looks like this:

$U \leftarrow A$
$L \leftarrow I$
$P \leftarrow I$
for $j \leftarrow 0$ **to** $\text{len}(A) - 2$ **do**
  choose $k \geqslant j$ such that $|U[k, j]|$ is maximised.
  $U[j, j:] \leftrightarrow U[k, j:]$
  $L[j, :j] \leftrightarrow L[k, :j]$
  $P[j, :] \leftrightarrow P[k, :]$
  for $i \leftarrow j + 1$ **to** $\text{len}(A) - 1$ **do**
    $L[i, j] \leftarrow U[i, j]/U[j, j]$
    $U[i, :] \leftarrow U[i, :] - L[i, j] \times U[j, :]$

This algorithm is known as partial pivotting in contrast with full pivotting in which both rows and columns are interchanged. It should also be noted that this algorithm is mathematically correct but there are more efficient versions in which less copying about takes place.

Exercise 7.8. * Modify your `factorise_lu(A)` function to include partial pivoting and return the permutation matrix P as well as L and U. Modify `solve_lu(A,b)` to employ your new function.

7.10 The official version

It's important to understand how matrix equations work, but efficient implementation of solution strategies is a black art all of its own. Numpy comes with efficient and stable implementations of many important linear algebra algorithms. In particular, matrix solution is implemented by `numpy.linalg.solve`. The eigenvalue routine `numpy.linalg.eig` is also particularly useful for practical work.

There is an even more complete set of linear algebra routines in the module `scipy.linalg` including `scipy.linalg.lu_factor` and `scipy.linalg.lu_solve` which form an implementation of the algorithm studied here.

7.11 Further reading

An excellent reference on a wide range of numerical linear algebra is Trefethen and Bau "Numerical Linear Algebra", SIAM, 1997. Lecture 20 covers Gaussian elimination and the LU factorisation.

Chapter 8

Curve fitting and least squares

8.1 Orthogonality

In order to understand the main subject of this lecture, we first need to revisit the concept of orthogonality. Orthogonality can be thought of as the property of being at right angles. *Perpendicular* and *normal* are other words which are used to describe orthogonality. From Maths Methods 1, you already know the relationship between the orthogonality of vectors and the dot product:

Definition 8.1. Two vectors $\mathbf{a}$ and $\mathbf{b}$ are *orthogonal* if and only if:

$$\mathbf{a} \cdot \mathbf{b} = 0 \tag{8.1}$$

Using what we know about the relationship between the dot product and matrix multiplication, we can draw some immediate conclusions from this definition:

Theorem 8.2. A vector $\mathbf{b}$ is *orthogonal* to all of the rows of a matrix A if and only if:

$$\mathrm{A}\mathbf{b} = \mathbf{0} \tag{8.2}$$

Theorem 8.3. A vector $\mathbf{b}$ is *orthogonal* to all of the columns of a matrix A if and only if:

$$\mathbf{b}\mathrm{A} = \mathbf{0} \tag{8.3}$$

8.2 Least squares solutions in two dimensions

In chapter 7 we learned that the span of one vector is the line passing through that vector. We also learned that if A has just one column there is no solution to the problem:

$$\mathrm{A}\mathbf{x} = \mathbf{b} \tag{8.4}$$

unless $\mathbf{b}$ is a multiple of the one column of A. Note that if A has only one column, $\mathbf{x}$ only has one entry. However we will continue to write it as a vector because we will later generalise this concept to more dimensions.

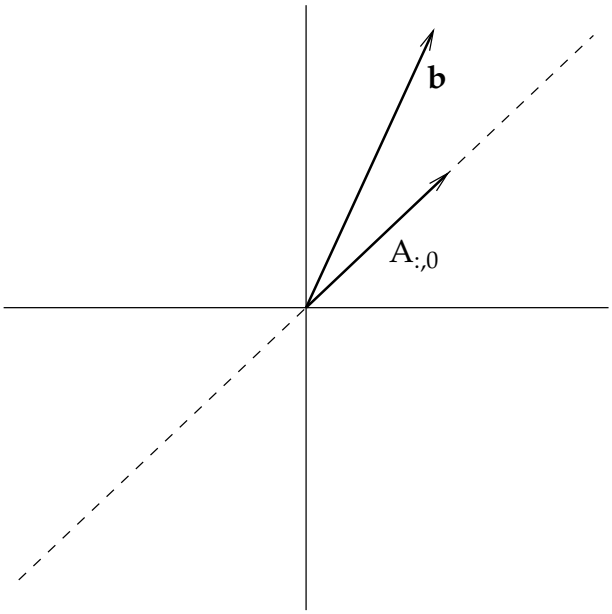

Figure 8.1: $A\mathbf{x} = \mathbf{b}$ has no solution if A has just one column and $\mathbf{b}$ is not a scalar multiple of that column.

The situation described by equation (8.4) is shown in figure 8.1 Clearly there is no multiple of A ($= A_{:,0}$) which will give $\mathbf{b}$. However it's also clear that some multiples of A are closer to $\mathbf{b}$ than others. To explore this concept we need to introduce the concept of a *residual*.

Definition 8.4. For a matrix equation:

$$A\mathbf{x} = \mathbf{b} \tag{8.5}$$

then for any choice of vector $\mathbf{x}$, the *residual* $\mathbf{r}$ is given by:

$$\mathbf{r} = \mathbf{b} - A\mathbf{x} \tag{8.6}$$

Note that if $\mathbf{x}$ is a solution to the equation, then the residual will be the zero vector, $\mathbf{0}$. The case where $\mathbf{x}$ is not the solution is illustrated in figure 8.2. The exciting thing about this is that it gives us a sense in which we can solve the problem $A\mathbf{x} = \mathbf{b}$. There is no exact solution, but we can find the solution for which the residual is the smallest possible.

What does it mean for the residual to be as small as possible? This is just saying that the length of $\mathbf{r}$ should be as small as possible. Remember that the length of $\mathbf{r}$ is given by $\sqrt{\mathbf{r} \cdot \mathbf{r}}$. So we want to minimise $\sqrt{\mathbf{r} \cdot \mathbf{r}}$. This is actually the same problem as minimising $\mathbf{r} \cdot \mathbf{r}$ or in other words $\mathbf{r}^2$, which is what gives this form of solution its name: *least squares*.

So, which vector $A\mathbf{x}$ will minimise the residual? It's possible to mathematically derive the answer, but actually it's obvious from thinking about the problem geometrically: $\mathbf{r} \cdot \mathbf{r}$ is minimised when $\mathbf{r}$ is *orthogonal* to $A\mathbf{x}$. Figure 8.3 illustrates this. So, when is $\mathbf{r}$ orthogonal to $A\mathbf{x}$? As we already know, $A\mathbf{x}$ always points in the same direction as

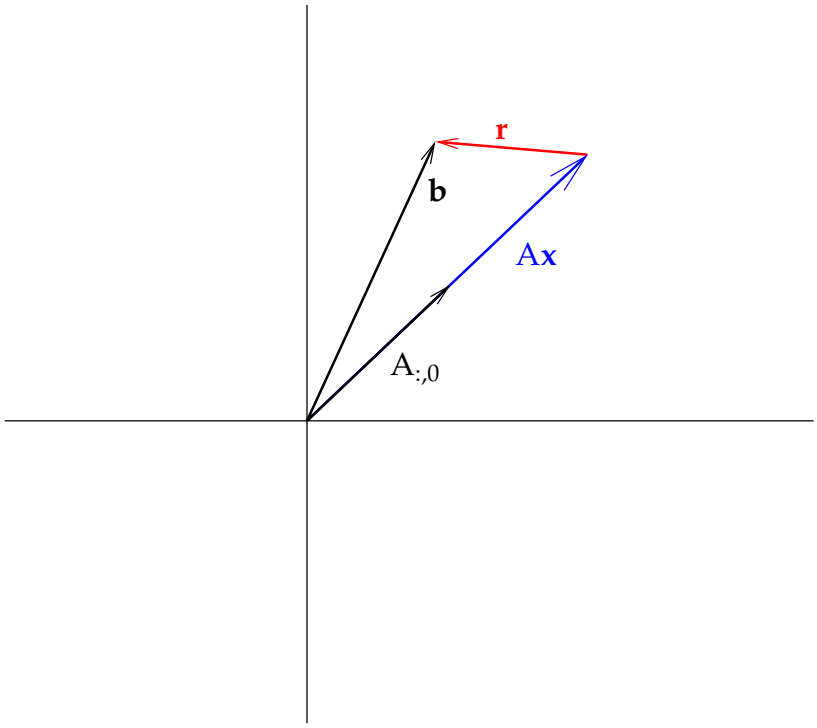

Figure 8.2: The residual $\mathbf{r} = \mathbf{b} - A\mathbf{x}$

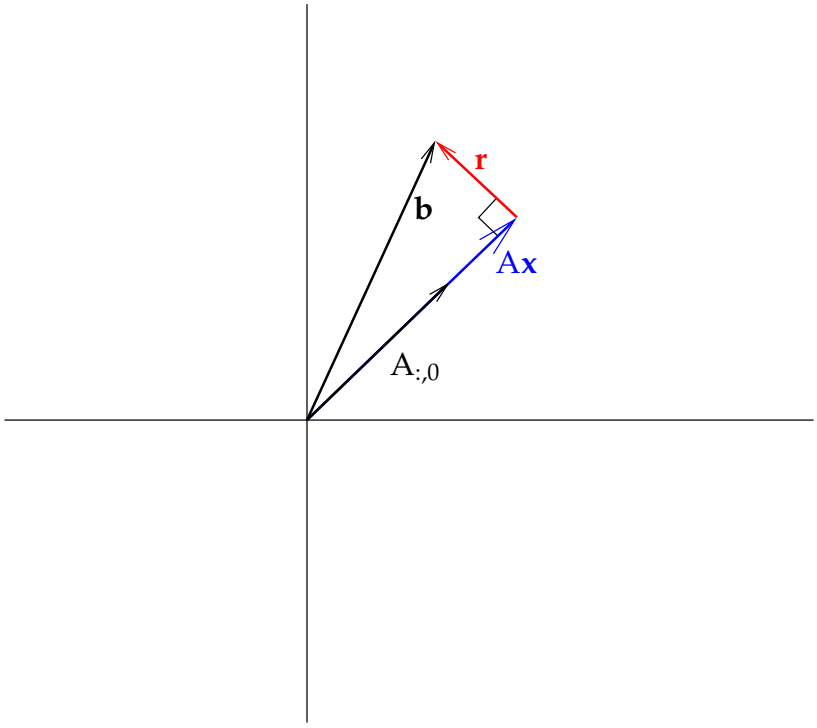

Figure 8.3: The residual is minimised when it is *orthogonal* to $A\mathbf{x}$

$A_{:,0}$. So $\mathbf{r}$ is orthogonal to $A\mathbf{x}$ when it is orthogonal to $A_{:,0}$. From the definition of orthogonality, we know this occurs when:

$$A_{:,0} \cdot \mathbf{r} = 0 \tag{8.7}$$

If we now substitute in the definition of $\mathbf{r}$, we get:

$$A_{:,0} \cdot (\mathbf{b} - A\mathbf{x}) = 0 \tag{8.8}$$

Now we can rearrange this equation:

$$A_{:,0} \cdot \mathbf{b} - A_{:,0} \cdot A\mathbf{x} = 0 \tag{8.9}$$
$$A_{:,0} \cdot A\mathbf{x} = A_{:,0} \cdot \mathbf{b} \tag{8.10}$$

Remember that A only has one column, so we can rewrite this as:

$$A_{:,0} \cdot A_{:,0}\mathbf{x} = A_{:,0} \cdot \mathbf{b} \tag{8.11}$$
$$|A_{:,0}|^2\mathbf{x} = A_{:,0} \cdot \mathbf{b} \tag{8.12}$$
$$\mathbf{x} = \frac{A_{:,0} \cdot \mathbf{b}}{|A_{:,0}|^2} \tag{8.13}$$

So that's it: we've solved this equation for $\mathbf{x}$.

8.2.1 2D least squares example

That was a bit complex so let's try a practical example. Suppose:

$$A = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{8.14}$$
$$\mathbf{b} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \tag{8.15}$$

and we want to solve:

$$A\mathbf{x} = \mathbf{b} \tag{8.16}$$

in a least squares sense.

Remember that since $\mathbf{A}$ has shape $(2, 1)$ and $\mathbf{b}$ has shape $(2,)$, $\mathbf{x}$ only has one entry and so has shape $(1,)$.

Using equation (8.13), we have:

$$\mathbf{x} = \frac{\begin{bmatrix} 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \end{bmatrix}}{\begin{bmatrix} 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix}} \tag{8.17}$$
$$= \frac{3}{2} \tag{8.18}$$

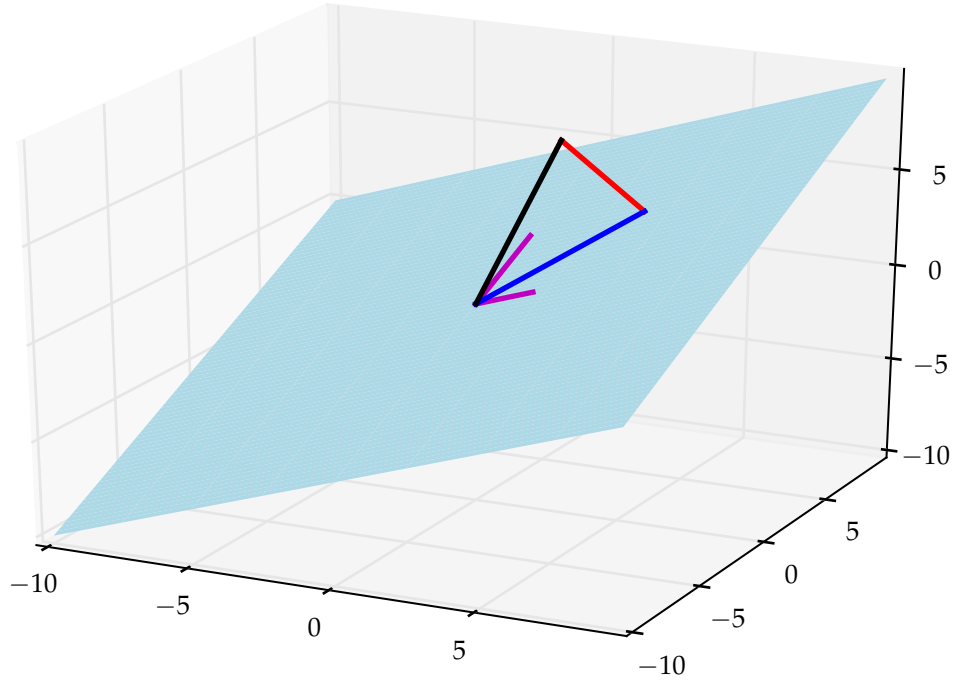

Figure 8.4: A three dimensional linear least squares problem in which A has two columns. The columns of A are shown in magenta, **b** is black, A**x** is blue and the residual, **r**, is red.

To see if this is correct, let's calculate A**x** and **r**:

$$\mathrm{A}\mathbf{x} = \begin{bmatrix} \frac{3}{2} \\ \frac{3}{2} \end{bmatrix} \tag{8.19}$$

$$\mathbf{r} = \mathbf{b} - \mathrm{A}\mathbf{x} \tag{8.20}$$

$$= \begin{bmatrix} 1 \\ 2 \end{bmatrix} - \begin{bmatrix} \frac{3}{2} \\ \frac{3}{2} \end{bmatrix} \tag{8.21}$$

$$= \begin{bmatrix} -\frac{1}{2} \\ \frac{1}{2} \end{bmatrix} \tag{8.22}$$

Note that **r** is orthogonal to $\mathrm{A}_{:,0}$:

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} -\frac{1}{2} \\ \frac{1}{2} \end{bmatrix} = 0 \tag{8.23}$$

8.3 Least squares in more dimensions

In more dimensions, A might have more columns but still not enough to span the space. For example in three dimensions, A might have two columns which are linearly

independent and the span of its columns would be a plane. In this case, A has shape $(3,2)$, $\mathbf{x}$ has shape $(2,)$ and $\mathbf{b}$ has shape $(3,)$.

Geometrically, $A\mathbf{x} = \mathbf{b}$ only has a solution if $\mathbf{b}$ lies in the plane spanned by the columns of A. There is always a least squares solution, however, and it is given by finding $\mathbf{x}$ such that the line from $\mathbf{b}$ to $A\mathbf{x}$ is orthogonal to the plane spanned by A. That line is, of course, the residual which is still given by:

$$\mathbf{r} = \mathbf{b} - A\mathbf{x} \tag{8.24}$$

This time, the least squares solution is achieved when $\mathbf{r}$ is orthogonal to *all* the columns of A. Figure 8.4 illustrates this situation.

When A had a single column, we could look for a residual orthogonal to A by taking the dot product between the single column of A and $\mathbf{r}$ and setting this to 0. With more columns in A, we need to take the dot product between each of them and $\mathbf{r}$ and set all the results to zero. In other words, we need to multiply the matrix whose *rows* are the *columns* of A by $\mathbf{r}$. This is achieved by taking the *transpose* of A. The condition for the least squares solution is therefore:

$$A^T\mathbf{r} = \mathbf{0} \tag{8.25}$$

Substituting the definition of $\mathbf{r}$ gives:

$$A^T(\mathbf{b} - A\mathbf{x}) = \mathbf{0} \tag{8.26}$$

$$A^T\mathbf{b} - A^TA\mathbf{x} = \mathbf{0} \tag{8.27}$$

$$A^TA\mathbf{x} = A^T\mathbf{b} \tag{8.28}$$

As a side note, the Python syntax for the transpose of an array `A` is `A.T`.

To attempt to understand what we've just done, let's consider the size of A, $\mathbf{x}$ and $\mathbf{b}$. We assumed that A has more rows than columns so shape(A) is (m,n) with $m > n$. shape($\mathbf{x}$) is therefore (n) and shape($\mathbf{b}$) is (m). Figure 8.5 illustrates the shapes of the arrays in the problem. Now, A^T has shape (n,m), so A^TA has shape (n,n), as shown in figure 8.6.

Similarly, $A^T\mathbf{b}$ is a vector of length n. Our least squares problem is therefore a square $n \times n$ matrix problem:

$$(A^TA)\mathbf{x} = A^T\mathbf{b} \tag{8.29}$$

So long as the columns of A^TA are linearly independent, we know how to solve this problem.

We will state without proof (to avoid additional complication), the condition which ensures that we can solve this least squares problem:

Theorem 8.5. The matrix A^TA has linearly independent columns if and only if the matrix A has linearly independent columns.

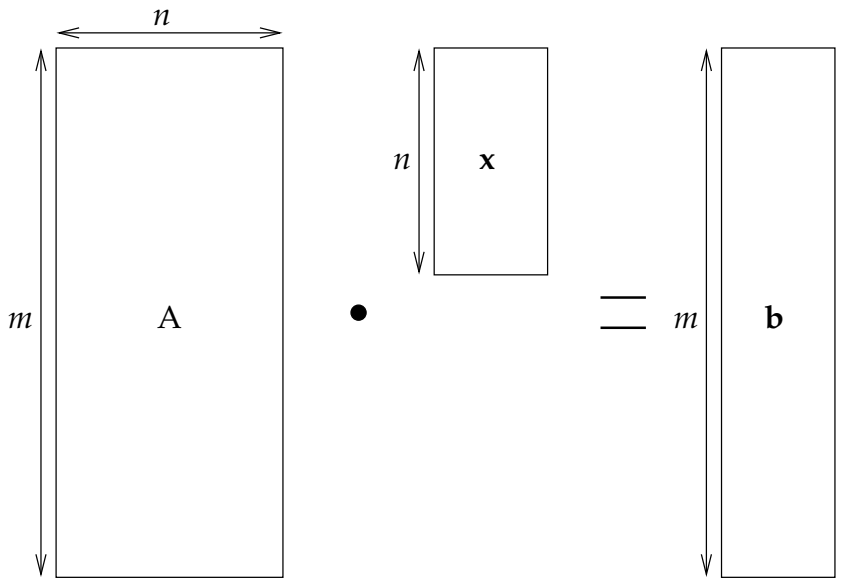

Figure 8.5: Matrix shapes in the overdetermined problem $A\mathbf{x} = \mathbf{b}$

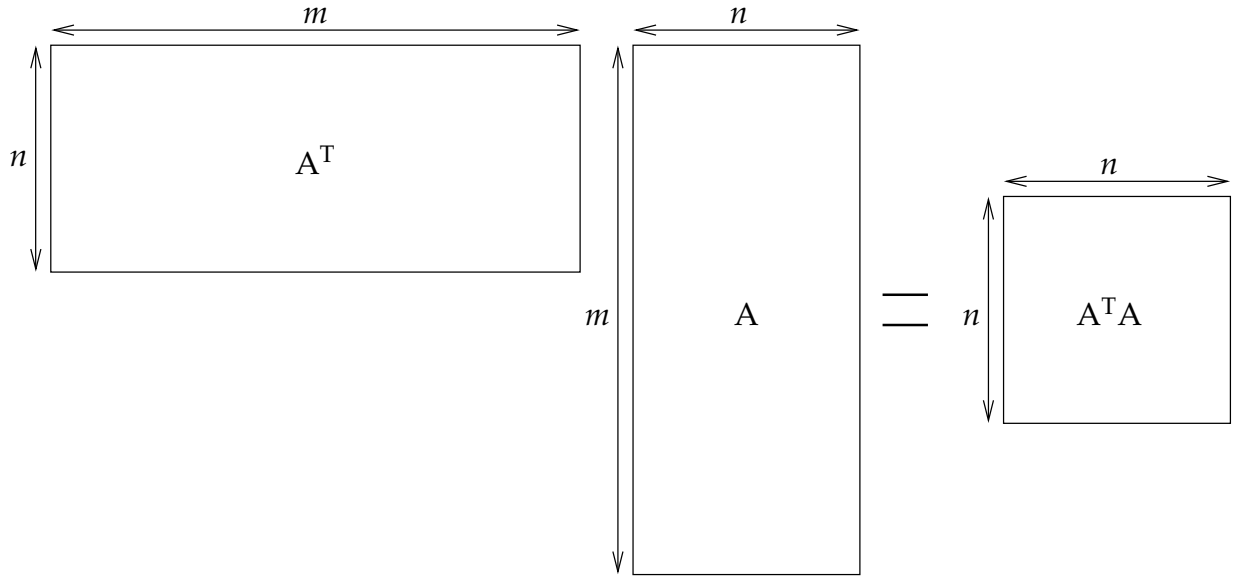

Figure 8.6: A^TA is a square $n \times n$ matrix.

8.3.1 3D Least squares example

To make this technique more concrete, let's try an example. Suppose

$$\mathrm{A} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \\ 0.5 & 1 \end{bmatrix} \tag{8.30}$$

$$\mathbf{b} = \begin{bmatrix} 2 \\ 2 \\ 8 \end{bmatrix} \tag{8.31}$$

Then:

$$\mathrm{A}^{\mathrm{T}}\mathrm{A} = \begin{bmatrix} 1.25 & 0.5 \\ 0.5 & 5 \end{bmatrix} \tag{8.32}$$

$$\mathrm{A}^{\mathrm{T}}b = \begin{bmatrix} 6 \\ 12 \end{bmatrix} \tag{8.33}$$

Solving $\mathrm{A}^{\mathrm{T}}\mathrm{A}\mathbf{x} = \mathrm{A}^{\mathrm{T}}\mathbf{b}$ gives:

$$\begin{bmatrix} 4 \\ 2 \end{bmatrix} \tag{8.34}$$

To check if this is correct, we can calculate the residual, $\mathbf{r}$:

$$\mathbf{r} = \mathbf{b} - \mathrm{A}\mathbf{x} \tag{8.35}$$

$$= \begin{bmatrix} 2 \\ 2 \\ 8 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 2 \\ 0.5 & 1 \end{bmatrix} \begin{bmatrix} 4 \\ 2 \end{bmatrix} \tag{8.36}$$

$$= \begin{bmatrix} 2 \\ 2 \\ 8 \end{bmatrix} - \begin{bmatrix} 4 \\ 4 \\ 4 \end{bmatrix} \tag{8.37}$$

$$= \begin{bmatrix} -2 \\ -2 \\ 4 \end{bmatrix} \tag{8.38}$$

Is this right? Well we can check by computing $\mathrm{A}^{\mathrm{T}}\mathbf{r}$:

$$\mathrm{A}^{\mathrm{T}}\mathbf{r} = \begin{bmatrix} 1 & 0 & 0.5 \\ 0 & 2 & 1 \end{bmatrix} \begin{bmatrix} -2 \\ -2 \\ 4 \end{bmatrix} \tag{8.39}$$

$$= \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{8.40}$$

We can complete all of these calculations easily in Python:

```
from numpy import *
A=array([[1.,0.],[0.,2.],[0.5,1.]])
b=array([2.,2.,8.])
```

```
x=numpy.linalg.solve(dot(A.T,A),dot(A.T,b))

r=b-dot(A,x)

# Least squares solution
print x
# Residual
print r
# Should be zero!
print dot(A.T,r)
```

Exercise 8.1. Given:

$$\mathrm{A} = \begin{bmatrix} 2 \\ 5 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} -2 \\ 9.5 \end{bmatrix} \tag{8.41}$$

Solve $\mathrm{A}\mathbf{x} = \mathbf{b}$ by hand using equation (8.13). Draw by hand a plot similar to figure 8.3 showing $\mathrm{A}_{:,0}$, $\mathrm{A}\mathbf{x}$, $\mathbf{b}$ and the residual $\mathbf{r}$.

Exercise 8.2. Write a Python script `test_least_squares.py`. The script should use `random_matrices.random_nonsquare` from `/numerical-methods-1`, generate random 3×2 and 3×4 matrices. Use `random_matrices.random_vec` to generate random right hand sides. Solve the resulting least squares systems and verify that your answer is correct by checking that the residual is orthogonal to the columns of A.

8.4 Curve fitting

8.4.1 Fitting a line through two points

Let's now consider one of the most common tasks in science. We have some experimental or observational data which we expect to fit some theoretical curve, but we don't know the right parameters. For example, suppose we have two input values (experimental machine settings, for example):

$$\mathbf{x} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \tag{8.42}$$

As a result of our experiment, we find two outcomes:

$$\mathbf{y} = \begin{bmatrix} 2 \\ 6 \end{bmatrix} \tag{8.43}$$

We want to fit a straight line of the form $y = c_0 x + c_1$ to these points. Using our two points, we can write two such equations:

$$c_0 x_0 + c_1 = y_0 \tag{8.44}$$
$$c_0 x_1 + c_1 = y_1 \tag{8.45}$$

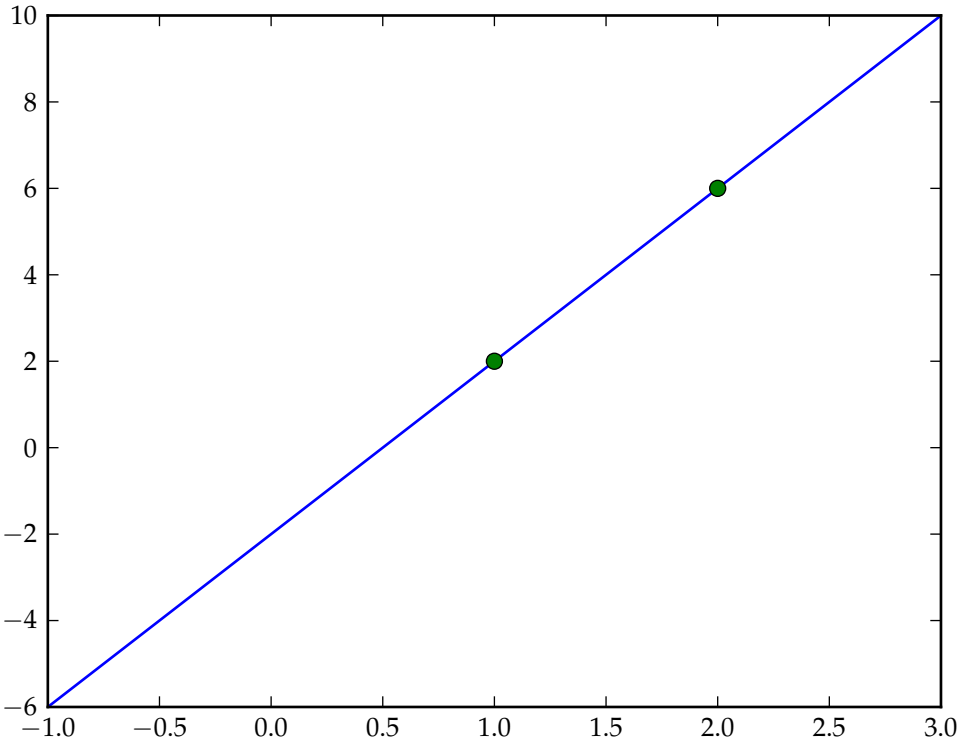

Figure 8.7: The line passing through the points $(1, 2)$ and $(2, 6)$

Of course we can also write this system of equations as a single matrix equation:

$$\begin{bmatrix} x_0 & 1 \\ x_1 & 1 \end{bmatrix} \mathbf{c} = \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} \tag{8.46}$$

By substituting our two points in, we achieve:

$$\begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix} \mathbf{c} = \begin{bmatrix} 2 \\ 6 \end{bmatrix} \tag{8.47}$$

By solving this, we find:

$$\mathbf{c} = \begin{bmatrix} 4 \\ -2 \end{bmatrix} \tag{8.48}$$

We can write a very easy Python code for this. The output of this program is figure 8.7.

```
from numpy import *
from pylab import *

def fit_linear(x,y):
    '''Construct and solve a linear system for the line
    passing through two points in the plane. x is the
    two x values, y is the two y values.

    The result is a vector c where c[0]x+c[1]=y is the line.
    '''

    if(x.shape!=(2,)):
        raise ValueError("x must be a 2-vector")
    if(y.shape!=(2,)):
        raise ValueError("y must be a 2-vector")

    A=zeros((2,2))

    A[:,0]=x
    A[:,1]=1.

    c=linalg.solve(A,y)

    return c

# x values to fit to.
x=array([1,2])
# y values to fit to.
y=array([2,6])

# solve for gradient and y intercept.
c=fit_linear(x,y)

# plot the original data.
plot(x,y,'o')

# plot the line we have fit.
X=linspace(-1,3)
Y=c[0]*X+c[1]

# plot x and y axes:
plot(X,0*X,'k')
plot(0*Y,Y,'k')

plot(X,Y)
axis([min(X),max(X),min(Y),max(Y)])

show()
```

8.4.2 Fitting higer degree polynomials

Suppose now we have *three* points through which we wish to fit a quadratic curve (a parabola). The process is essentially identical except now the general quadratic curve is written as $c_0x^2 + c_1x + c_2 = 0$. Suppose I have a vector of x values and a vector of corresponding y values:

$$\mathbf{x} = \begin{bmatrix} 1 \\ 3 \\ 4 \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} 6 \\ 18 \\ 27 \end{bmatrix} \tag{8.49}$$

This time, our matrix system is:

$$\left[\begin{array}{c|c|c} \mathbf{x}^2 & \mathbf{x} & 1. \end{array}\right] \mathbf{c} = \mathbf{y} \tag{8.50}$$

Substituting $\mathbf{x}$ and $\mathbf{y}$, we have:

$$\begin{bmatrix} 1. & 1. & 1. \\ 9. & 3. & 1. \\ 16. & 4. & 1. \end{bmatrix} \mathbf{c} = \begin{bmatrix} 6 \\ 18 \\ 27 \end{bmatrix} \tag{8.51}$$

For which the solution is:

$$\mathbf{c} = \begin{bmatrix} 1. \\ 2. \\ 3. \end{bmatrix} \tag{8.52}$$

The quadratic which fits this data is therefore $y = x^2 + 2x + 3$.

It is easy to see that we could extend this to fit a degree n polynomial through $n + 1$ points. A Python function which does this is:

```
def fit_polynomial(x,y, n):
  '''Construct and solve a linear system for the degree n
  polynomial passing through n+1 points in the plane.
  x is the n+1 x values, y is the n+1 y values.

  The result is a vector c where
            c[0]x^n+c[1]x^(n-1)+...+c[n]=y
  is the line.
  '''

  if(x.shape!=(n+1,)):
      raise ValueError("x must be a n+1-vector")
  if(y.shape!=(n+1,)):
      raise ValueError("y must be a n+1-vector")

  A=zeros((n+1,n+1))

  for i in range(n+1):
      A[:,i]=x**(n-i)

  return linalg.solve(A,y)
```

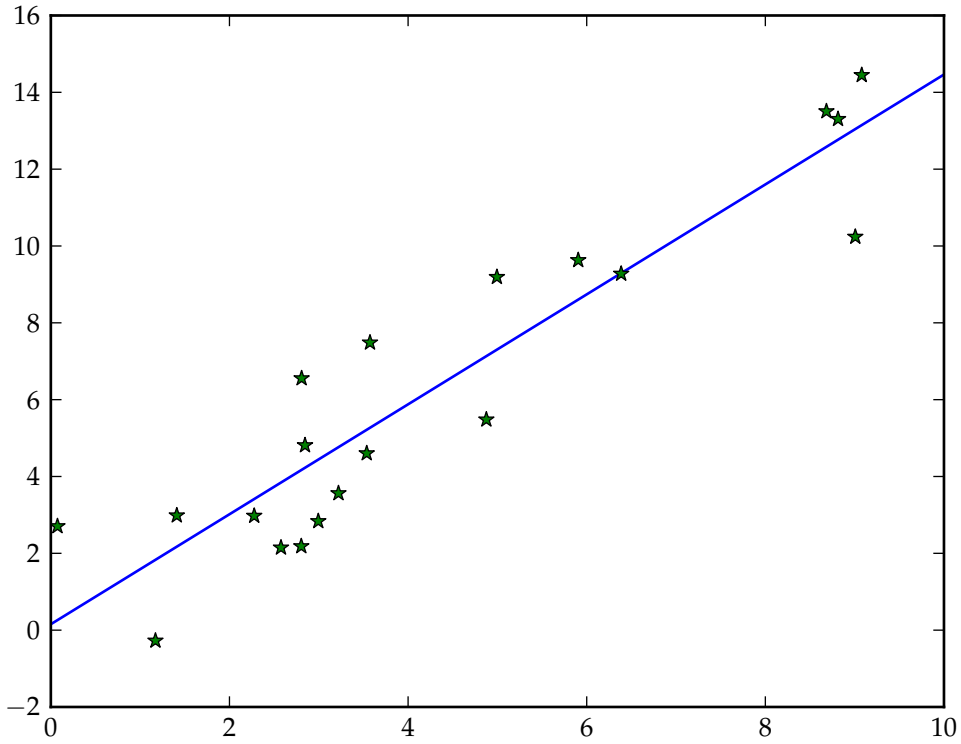

Figure 8.8: The linear fit to a set of data.

8.4.3 Overdetermined curve fitting

The scenario in which we have exactly $n+1$ points to which we wish to fit to a degree n polynomial is not typical in science. Instead, we usually have a large data set for which we have good theoretical reasons to believe that the data should lie on some low degree polynomial curve such as a straight line or a parabola.

Of course experimental data is usually noisy so the points won't line up exactly on the line. What we want is the line which best fits the set of points, in some sense. Figure 8.8 shows this situation.

So, how do we fit this line? Well we can first construct our linear system as before. Suppose we have a set of independent variable values $\mathbf{x}$ and a set of data $\mathbf{y}$. Then if they all lie on the same line, the equation is:

$$\left[\begin{array}{c|c} \mathbf{x} & 1. \end{array}\right] \mathbf{c} = \mathbf{y} \tag{8.53}$$

Of course this equation has many more equations than unknowns so unless we are absolutely, stupidly fortunate, there will be no solution. However we have just learned that a system like this can be solved in the least squares sense by multiplying on the

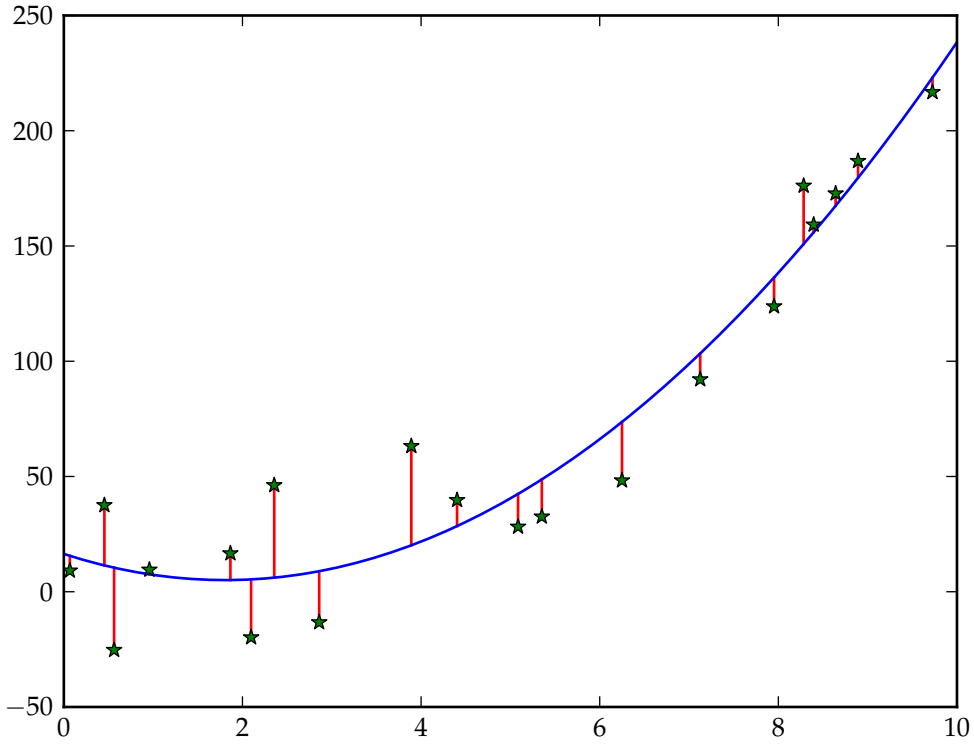

Figure 8.9: A data set with fitted parabola. The red vertical lines show the magnitude of the residual component associated with each data point.

left by the transpose of the matrix. The least squares system is:

$$A^T A\mathbf{c} = A^T\mathbf{y} \tag{8.54}$$

$$A = \left[\begin{array}{c|c} \mathbf{x} & 1. \end{array}\right] \tag{8.55}$$

This system is 2×2 and gives the best line through the points in some least squares sense.

Exactly the same argument for fitting a polynomial of higher degree to a dataset applies, only this time the matrix A will have columns corresponding to the higher powers of x.

8.4.4 Interpretation of the least squares solution

So what does the least squares solution mean for curve fitting? Remember that it is the solution which minimises the residual $\mathbf{r}$ given by:

$$\mathbf{r} = \mathbf{y} - A\mathbf{x} \tag{8.56}$$

In the curve fitting case, the points A**x** lie on the curve so the residual is the distance from the curve to the data points **y** in the vertical direction. Figure 8.9 shows this distance for each point in a data set with its least squares quadratic fit.

It would be convenient to have a single number which indicates how well the curve fits the data. The obvious candidate is the magnitude of the residual:

$$|\mathbf{r}| = \sqrt{\mathbf{r} \cdot \mathbf{r}} \tag{8.57}$$

$$= \sqrt{\sum_i r_i^2} \tag{8.58}$$

The issue with this quantity is that it always increases as more data is added, which is not a welcome feature of an error indicator. To take into account the size of the data set, we can use the *root mean square error*. This is given by:

$$\text{RMS}(\mathbf{r}) = \sqrt{\frac{\sum_i r_i^2}{n}} \tag{8.59}$$

where n is the size of the data set and therefore the length of $\mathbf{r}$.

If you think back to secondary school statistics, this formula looks very familiar. In fact it's almost the same as the standard deviation formula. If we consider the curve as some sort of average for the data, then the RMS error is the standard deviation of the difference between the data value and the curve value for a given x.

Exercise 8.3. Produce a modified version of `fit_polynomial` from page 82 which performs a least squares fit for any number of data points greater than the polynomial degree. The original `fit_polynomial` is available from `/numerical-methods-1/fit_polynomial.py`. Use `fit_polynomial.random_data` to test your routine and produce plots similar to figure 8.8 for at least linear and quadratic input data. *Hint:* the checks for incorrect input to `fit_polynomial` will need to change. What checks apply to the least squares version?

8.5 Official versions

As with matrix solution, `numpy` comes with least squares solvers and a special purpose routine for curve fitting which are likely to be faster and more robust than anything you write for yourself. The `numpy` function for least squares systems is `numpy.linalg.lstsq` while the one for polynomial curve fitting is `numpy.polyfit`. Another useful function in this regard is `numpy.poly1d` which takes a sequence of coefficients and returns the corresponding polynomial function. This function can then be called on x values to evaluate the polynomial.

Appendix A

Taylor series

A.1 Power series

A power series is a function represented by an infinite sum:

$$\begin{aligned} f(x) &= a_0 + a_1(x - x_0) + a_2(x - x_0)^2 + \ldots + a_n(x - x_0)^n + \ldots \\ &= \sum_{k=0}^{\infty} a_k(x - x_0)^k \end{aligned} \tag{A.1}$$

The point x_0, often referred to as the *centre* of the power series, is fixed: it does not depend on x. Similarly, the coefficients a_k are independent of x.

For the purposes of numerical mathematics, it is most convenient to write power series in another form. By making the substitution $x = x_0 + h$, we can write:

$$f(x_0 + h) = \sum_{k=0}^{\infty} a_0 h^k \tag{A.2}$$

A.2 Radius of convergence

A power series representation of f is only valid for values of h for which the series converges. It is possible to test for the convergence of a series using the ratio test, otherwise known as d'Alembert's test after the 18th century French mathematician Jean de Rond d'Alembert who originally published it. The test establishes the following sufficient test for convergence:

$$\lim_{n\to\infty} \left| \frac{a_{n+1} h^{n+1}}{a_n h^n} \right| < 1 \tag{A.3}$$

or equivalently:

$$|h| \lim_{n\to\infty} \left| \frac{a_{n+1}}{a_n} \right| < 1 \tag{A.4}$$

Hence the convergence of the power series depends not only on the coefficients a_n, it is also linear in the magnitude of h. This enables us to define a new parameter R, the radius of convergence. The convergence criteria now becomes:

$$|h| < R \quad \Longleftrightarrow \quad x_0 - R < x < x_0 + R \tag{A.5}$$

where:

$$R \equiv \lim_{n\to\infty} \left| \frac{a_n}{a_{n+1}} \right| \tag{A.6}$$

A power series *centred* at x_0 is therefore valid for values h less than the radius R.

The radius of convergence of a power series is preserved under addition, subtraction, multiplication and differentiation. That is to say, if two power series with the same radius of convergence are added, subtracted or multiplied then the resulting power series has the same radius of convergence. Similarly, the derivative of a power series is a power series with the same radius of convergence.

A.3 Taylor and MacLaurin Series

Suppose we have some function $f(x)$ whose derivatives we know at some point x_0. We can write:

$$f(x_0 + h) = a_0 + a_1 h + a_2 h^2 + \ldots + a_n h^n + \ldots \tag{A.7}$$

If we set $h = 0$ then we note that:

$$f(x_0) = a_0 \tag{A.8}$$

Differentiating the series we have:

$$f'(x) = a_1 + 2a_2 h + 3a_3 h^2 \ldots + n a_n h^{n-1} + \ldots \tag{A.9}$$

so that:

$$f'(x_0) = a_1 \tag{A.10}$$

If we differentiate again, we have:

$$f''(x) = 2a_2 + 2 \times 3a_3 h \ldots + (n-1) \times n a_n h^{n-2} + \ldots \tag{A.11}$$

yielding:

$$f''(x_0) = 2a_2 \tag{A.12}$$

similarly:

$$\begin{aligned} f'''(x_0) &= 3 \times 2a_3 \\ &= 3!\, a_3 \end{aligned} \tag{A.13}$$

In general:

$$f^{(n)}(x_0) = n!\, a_n \tag{A.14}$$

In other words:

$$a_n = \frac{f^{(n)}(x_0)}{n!} \tag{A.15}$$

This gives us a new way of representing a function:

$$\begin{aligned} f(x_0+h) &= f(x_0) + f'(x_0)h + \frac{f''(x_0)}{2!}h^2 + \ldots + \frac{f^{(n)}(x_0)}{n!}h^n + \ldots \\ &= \sum_{k=0}^{\infty} \frac{f^{(k)}(x_0)}{k!}h^k \end{aligned} \tag{A.16}$$

provided, of course, that the series converges. In this final formula we employ the conventions that the zeroth derivative of a function is the function itself ($f^{(0)}(x) = f(x)$) and that $0! = 1$.

This means we have an algorithm for expressing $f(x)$ as a power series subject to the radius of convergence and assuming that the derivatives $f^{(n)}(x)$ are continuous within that radius of convergence.

This form of power series is known as a Taylor series. A special case of the Taylor series is achieved by setting $x_0 = 0$:

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(0)}{k!}x^k \tag{A.17}$$

This form of Taylor series is referred to as a MacLaurin series.

A.3.1 Example: e^x

Expand $f(x) = e^x$ around $x = 0$.

e^x is unique in that:

$$f(x) = f'(x) = f''(x) = f^{(n)}(x) = e^x \tag{A.18}$$

so at $x = 0$ all the derivatives are $e^0 = 1$. This produces the series:

$$\begin{aligned} e^x &= \sum_{k=0}^{\infty} \frac{x^k}{k!} \\ &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots \end{aligned} \tag{A.19}$$

The radius of convergence for this series is:

$$\begin{aligned} R &= \lim_{n\to\infty} \left| \frac{a_n}{a_n + 1} \right| \\ &= \lim_{n\to\infty} \left| \frac{(n+1)!}{n!} \right| \\ &= \lim_{n\to\infty} n + 1 \\ &= \infty \end{aligned} \tag{A.20}$$

That is to say, the expansion converges for any x.

A.3.2 Example: $\cos x$

Expand $f(x) = \cos x$ around $x = 0$.

In this case we have:

$$\begin{aligned} f'(x) &= -\sin x & f''(x) &= -\cos x \\ f'''(x) &= \sin x & f''''(x) &= \cos x \\ f^{(5)}(x) &= -\sin x & f^{(6)}(x) &= -\cos x \\ f^{(7)}(x) &= \sin x & f^{(8)}(x) &= \cos x \end{aligned}$$

So at $x = 0$ all the odd derivatives are 0 while the even derivatives alternate between -1 and 1. This produces:

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots \tag{A.21}$$

once again, the radius of convergence is ∞. By a similar process:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \tag{A.22}$$

A.3.3 Example: $\ln x$

Since $\ln x$ is not defined at $x = 0$, there is no MacLaurin series for this function. Instead, we produce a Taylor series for $\ln x$ around $x_0 = 1$.

In this case we have the following derivatives:

$$\begin{aligned} f'(x) &= \frac{1}{x} \\ f''(x) &= -\frac{1}{x^2} \\ f'''(x) &= \frac{2}{x^3} \\ f''''(x) &= \frac{3 \times 2}{x^4} \\ f^{(n)}(x) &= -1^{n-1}\frac{(n-1)!}{x^n} \end{aligned}$$

So that at $x = 1$ we have:

$$\begin{aligned} f(1) &= 0 \\ f'(1) &= 1 \\ f''(1) &= -1 \\ f'''(1) &= 2 \\ f''''(1) &= -6 \\ &\vdots \end{aligned}$$

This leads us to:

$$\begin{aligned} \ln(1+h) &= \sum_{k=1}^{\infty} -1^{k-1} \frac{(k-1)!}{k!} h^k \\ &= \sum_{k=1}^{\infty} -1^{k-1} \frac{h^k}{k} \\ &= h - \frac{h^2}{2} + \frac{h^3}{3} + \dots \end{aligned} \tag{A.23}$$

This has radius of convergence:

$$\begin{aligned} R &= \lim_{n\to\infty} \left| \frac{a_n}{a_{n+1}} \right| \\ &= \lim_{n\to\infty} \left| \frac{n+1}{n} \right| \\ &= 1 \end{aligned} \tag{A.24}$$

So the series converges for $-1 < h < 1$ or $0 < x < 2$.