MA934 - Numerical Algorithms and Optimisation

Dr. Radu Cimpeanu
with thanks to Prof. Colm Connaughton
Mathematical Modelling for Real-World Systems CDT

October 24, 2022

Contents

Notes 1: Finite precision arithmetic, algorithms and computational complexity

1.1 Numerical arithmetic and precision

This module is about using digital computers to do calculations and process data. As a prelude it is worth learning a little bit about how digital computers do arithmetic because all is not always as it seems. Arithmetic is here taken to mean the operations of addition/subtraction and multiplication/division.

Let us begin by asking a simple question: what is a number? In the familiar (decimal) representation of integers, the string of digits $d_2d_1d_0$ is really shorthand for

$$d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0,$$

where the digits, $d_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Likewise a real number, having decimal representation $d_2d_1d_0.d_{-1}d_{-2}d_{-3}$, is shorthand for

$$d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0 + d_{-1} \times 10^{-1} + d_{-2} \times 10^{-2} + d_{-3} \times 10^{-3}.$$

There is no fundamental reason why we should work in base 10. In a base-b representation, the number $(d_2d_1d_0.d_{-1}d_{-2}d_{-3})_b$ would be shorthand for

$$d_2 \times b^2 + d_1 \times b^1 + d_0 \times b^0 + d_{-1} \times b^{-1} + d_{-2} \times b^{-2} + d_{-3} \times b^{-3},$$

where the digits $d_i \in \{0, 1, 2, \ldots b-1\}$.

Digital computers represent numbers using base 2. A single binary digit is called a "bit". Physically, a bit is an electrical circuit which can be off or on thereby representing the two binary digits, 0 and 1. A string of 8 bits is called a "byte". Binary arithmetic, although convenient for computers, is very unwieldy for humans. Hence the nerdy joke: there are 10 types of people in the world - those who understand binary arithmetic and those who don't.

Numbers are represented in a computer by a string of bits of a fixed length. Most commonly used are 32-bit and 64-bit representations. As a result of this fixed length, the set of possible numbers is finite. At this level, we see that computer arithmetic cannot be equivalent to conventional arithmetic.

1.1.1 Representation of integers

Setting aside this finiteness, integers can be represented exactly provided they are not too large. For example an unsigned n-bit integer is represented as $x = (d_{n-1} \ldots d_1 d_0)_2$ with $d_i \in \{0, 1\}$. The largest and smallest integers representable in this format are

$$x_{max} = \sum_{i=0}^{n-1} 2^i = 2^n - 1 \qquad x_{min} = 0.$$

To represent signed integers, we can use one of the bits to encode the sign.

1.1.2 Representation of real numbers - fixed and floating point

Real numbers generally require an infinite string of digits and so cannot be represented exactly using a finite string of bits. Computers therefore simulate real arithmetic. Note that numbers which have a terminating expansion for one choice of base may have a non-terminating expansion in another. For example $(0.1)_{10} = (0.0001100110011001100110011 \ldots)_2$. One approach is to store a fixed number of integer and fractional digits. This is called a fixed-point representation. For example, we can use $2n$ bits to represent a real number with the following signed fixed-point representation:

$$x = \pm(d_{n-1} \ldots d_1 d_0 . d_{-1} d_{-2} \ldots d_{-n+1})_2,$$

with $d_i \in \{0, 1\}$. We necessarily introduce an error in representing numbers whose binary expansions contain more fractional digits than $n-1$. This error is called round-off error. It is a feature of hardware and cannot be avoided. Hardware can implement different protocols for rounding:

- Round to zero
- Round half away from zero
- Round half to even

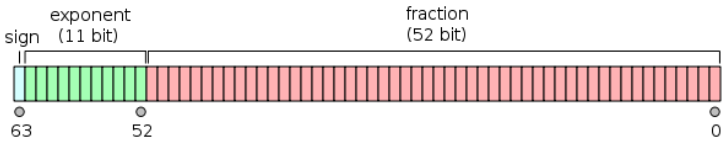
The advantage of fixed point representation is that the representable numbers are uniformly spaced. The big disadvantage is that they cannot represent very big or very small numbers as commonly arise in science and engineering. For this latter reason, fixed point representations are rarely if ever used in practice. A better approach is the so-called floating-point representation in which a number consists of two parts:

- a significand: contains the number's digits. Negative numbers have negative significands.
- an exponent: says where the decimal (or binary) point is placed relative to the beginning of the significand. Negative exponents represent numbers that are very small (i.e. close to zero).

IEEE 754 provides the standard for floating point representation of real numbers. The standard defines a 32-bit format known as a single precision and a 64 bit format known as double precision. The bits are used as follows:

Format	Total bits	Significand bits	Exponent bits
single precision	32	23 + sign	8
single precision	64	52 + sign	11

In memory, a double precision number looks like the following:



This format has the attractive properties:

- it can represent numbers having very different magnitudes. This is limited by the length of the exponent.
- the relative accuracy is the same at all magnitudes. This is limited by the length of the significand. The absolute accuracy depends on the scale. The representable numbers are not uniformly spaced.
- multiplying a very large and a very small number preserves the accuracy of both in the result.

For more detail than you every wanted to know about floating point arithmetic see [1] and the references therein.

1.1.3 Rounding errors

Rounding errors mean that floating point arithmetic can have properties which seem bizarre if one forgets the fact that its purpose is to simulate conventional arithmetic, not to reproduce it. Consider for example the following piece of C code:

```
double a,b,c;
a=0.1;
b=0.2;
c=0.1+0.2;
fprintf(stdout, "a=%.16e\n",a);
fprintf(stdout, "b=%.16e\n",b);
fprintf(stdout, "c=%.16e\n",c);
```

which gives the output

```
research@cleo:~$ ./a.out
a=1.0000000000000001e-01
b=2.0000000000000001e-01
c=3.0000000000000004e-01
```

Round-off error can mean that statements which seem mathematically correct can be false. Here is another example:

```
double a,b;
a=0.15 + 0.15;
b=0.1+0.2;
if(a==b)
{fprintf(stdout, "True\n");}
else
{fprintf(stdout, "False\n");}
```

which gives the output

```
research@smudge:~$ ./a.out
False
```

The rounding errors in these examples are small. The machine accuracy, ϵ_m, is the smallest floating point number which when added to the floating point number 1.0 produces a result which is different from 1.0. IEEE 754 single precision format has $\epsilon_m \approx 1.19 \times 10^{-7}$ whereas IEEE 754 double precision format has $\epsilon_m \approx 2.22 \times 10^{-16}$. Any operation on floating point numbers should be thought of as introducing an error of at least ϵ_m. When the results of such operations are fed into other operations to form an algorithm, these errors propagate through the calculations. One might hope that random errors of opposite sign cancel each other so that the error in an n-step calculation scales as $\epsilon \sim \sqrt{n}\,\epsilon_m$. In practice,

- some algorithms contain regularities which mean that $\epsilon \sim n\,\epsilon_m$.
- certain individual floating point operations can hugely amplify the error.

A particularly important example of the latter is known as "loss of significance". This occurs when two nearly equal numbers are subtracted. Loss of significance is best illustrated with an example from finite precision decimal arithmetic. Let us represent numbers, x, to 5-digit decimal precision with the notation $\mathrm{fl}(x)$. Consider the following exact arithmetic calculation:

$$x = 0.123456789 \qquad y = 0.1234 \qquad z = x - y = 0.000056789.$$

Repeating the calculation in 5-digit precision gives

$$\mathrm{fl}(x) - \mathrm{fl}(y) = 0.12345 - 0.12340 = 0.00005.$$

Note that the result contains only a single significant digit. This has terrible consequences for the relative accuracy of the calculation. Although the relative size of the rounding errors in x and y are very small

$$\frac{x - \mathrm{fl}(x)}{x} \approx 0.0000549 \qquad \frac{y - \mathrm{fl}(y)}{y} = 0,$$

the relative size of the rounding error in $x - y$ is almost 12%:

$$\frac{z - (\mathrm{fl}(x) - \mathrm{fl}(y))}{z} \approx 0.1195!$$

1.1.4 Stability of algorithms

Certain operations can amplify rounding errors. This can sometimes lead to catastrophic failure when algorithms which are exact in conventional arithmetic are executed in floating point. Such algorithms are said to be numerically unstable. The branch of mathematics which studies how errors propagate in algorithms is called numerical analysis. Although somewhat artificial, the following example from [2, chap. 1] clearly illustrates the concept of numerical instability. Consider the problem of calculating the series, $\{a_n = \phi^n,\ n = 0, 1, 2, \ldots\}$, of integer powers of the golden mean, $\phi = (\sqrt{5} - 1)/2$. This can be done in two ways:

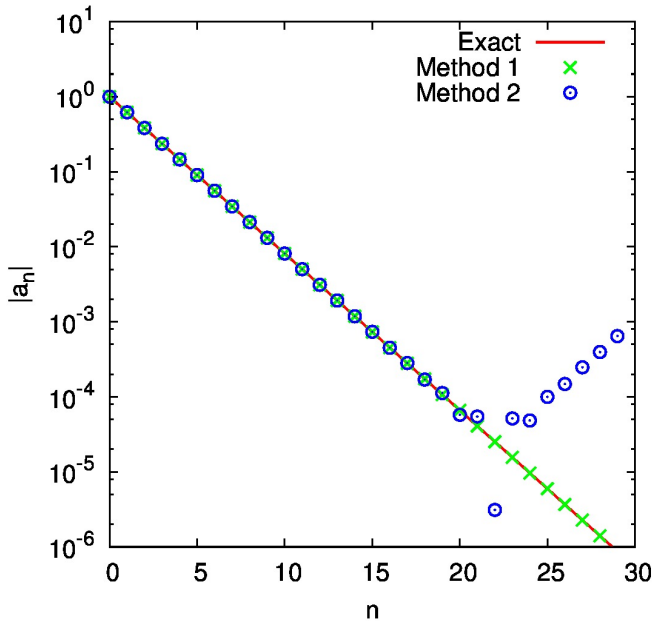


Figure 1.1: Numerical implementation of algorithms (1.1) and (1.2)

- **Method 1:**
  Set $a_0 = 1$ and calculate
$$a_n = \phi\, a_{n-1} \qquad \text{for } n = 1, 2, \ldots \tag{1.1}$$
- **Method 2:**
  Set $a_0 = 1$, $a_1 = \phi$ and calculate
$$a_n = a_{n-2} - a_{n-1} \qquad \text{for } n = 2, 3, \ldots \tag{1.2}$$

The second method is less obvious but it is easily demonstrated by direct substitution that ϕ satisfies this recurrence relation. Both are mathematically equivalent and correct. Fig. 1.1 shows the implementation of these two methods on my laptop computer with $\phi = 0.61803399$ (8 digits of precision). Clearly method 2 starts to go seriously wrong by about $n = 20$ and gives entirely wrong results for larger values of n. The reason is that the algorithm (1.2) is unstable and amplifies the rounding error in the value of ϕ.

The reason for this can be seen by noting that the computer is not solving Eq. (1.2) with the starting conditions $a_0 = 1$ and $a_1 = \phi$ but rather with the starting conditions $a_0 = 1$ and $a_1 = \phi + \varepsilon$. If we make the ansatz
$$a_n = x^n,$$
we find by direct substitution that Eq. (1.2) is satisfied provided that
$$x = x_+ \qquad \text{or} \qquad x = x_-$$
where
$$x_\pm = \frac{-1 \pm \sqrt{5}}{2}.$$
$x_+ = \phi$ corresponds to the solution we want. Since Eq. (1.2) is a linear recurrence relation, the general solution is
$$a_n = C_1\, x_+^n + C_2\, x_-^n,$$
where $C_{1,2}$ are constants. If we use the starting conditions $a_0 = 1$ and $a_1 = \phi + \varepsilon$ to evaluate these constants we obtain
$$a_n = \left(1 - \frac{\varepsilon}{\sqrt{5}}\right) \phi^n + \frac{\varepsilon}{\sqrt{5}}\, x_-^n.$$

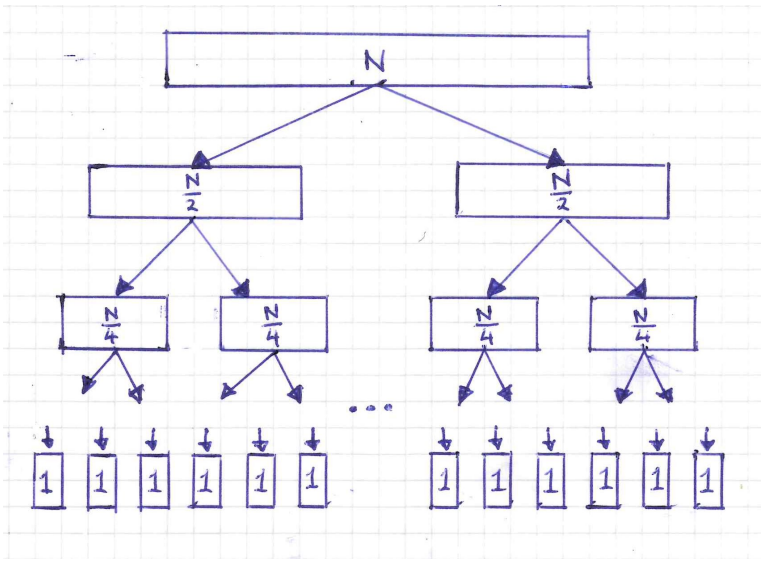


Figure 1.2: Conceptualisation of the recursive structure of a divide-and-conquer algorithm.

The second term grows (in a sign-alternating fashion) with n. Therefore any small amount of error, ε, introduced by finite precision is amplified by the iteration of the recurrence relation and eventually dominates the solution. Avoiding such instabilities is a key priority in designing good numerical algorithms.

1.2 Computational complexity of algorithms

1.2.1 Counting FLOPS and Big O notation

The performance speed of a computer is usually characterised by the number of FLoating-point OPerations per Second (FLOPS) which it is capable of executing. Computational complexity of an algorithm is characterised by the rate at which the total number of arithmetic operations required to solve a problem of size n grows with n. We shall denote this number of operations by $F(n)$. The more efficient the algorithm, the slower the $F(n)$ grows with n. The computational complexity of an algorithm is usually quantified by quoting an asymptotic bound on the behaviour of $F(n)$ as $n \to \infty$. The common notation, $F(n) \sim O(g(n))$, means that for sufficiently large values of n,

$$|F(n)|\,| \leq k \cdot g(n)$$

for some positive k. This says that as n gets large, the absolute value of $F(n)$ is bounded from above by a positive multiple of the function $g(n)$. There is a proliferation of more precise notation in use to express lower bounds and degrees of tightness of bounds which we will not really need here but which you can read about on Wikipedia [3]

Questions of computational complexity and deciding whether a particular algorithm is optimal or not can by highly nontrivial. For example, consider the problem of multiplying two matrices of size $n \times n$:

$$\mathbf{A}\,\mathbf{B} = \mathbf{C}.$$

If we use textbook matrix multiplication, we must compute n^2 entries to arrive at the matrix $\mathbf{C}$. Each of these entries requires taking the scalar product of a row from $\mathbf{A}$ with a column from $\mathbf{B}$. Since each is of length n, the scalar product requires n multiplications and n additions, or $2n$ FLOPs. The total number of operations required to multiply the two matrices is then $n^2 \times 2n = 2n^3$. So textbook matrix multiplication is a $O(n^3)$ algorithm.

1.2.2 Algorithmic complexity of matrix multiplication and Strassen's algorithm

It is interesting to ask could we do any better than $2n^3$? Suppose for simplicity that $n = 2^m$ for some integer m. We could then go about the multiplication differently by dividing the matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$

into smaller matrices of half the size,

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{pmatrix} \quad \mathbf{C} = \begin{pmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{pmatrix}$$

up the problem into smaller matrix multiplications of half the size:

$$\begin{aligned} \mathbf{C}_{11} &= \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} \\ \mathbf{C}_{12} &= \mathbf{A}_{11}\mathbf{B}_{21} + \mathbf{A}_{12}\mathbf{B}_{22} \\ \mathbf{C}_{21} &= \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} \\ \mathbf{C}_{22} &= \mathbf{A}_{21}\mathbf{B}_{21} + \mathbf{A}_{22}\mathbf{B}_{22}. \end{aligned} \tag{1.3}$$

This is an example of a type of recursive algorithm known as a "divide and conquer" algorithm. The operation of multiplication of matrices of size $n \times n$ is defined in terms of the multiplication of matrices of size $\frac{n}{2} \times \frac{n}{2}$. It is implicitly understood in Eq. (1.3) that when we get to matrices of size 1×1, we do not recurse any further but simply use scalar multiplication (this is called the "base case" in the literature on recursive algorithms). The conceptual structure of a divide-and-conquer algorithm is illustrated in Fig. 1.2.It may not be immediately clear whether we gained anything by reformulating the textbook matrix multiplication algorithm in this way. Eq. (1.3) involves 8 matrix multiplications of size $\frac{n}{2}$ and 4 matrix additions of size $\frac{n}{2}$ (matrix addition clearly requires n^2 FLOPs). Therefore the computational complexity, $F(n)$, must satisfy the recursion

$$F(n) = 8\,F\left(\frac{n}{2}\right) + 4\,\left(\frac{n}{2}\right)^2,$$

with the starting condition $F(1) = 1$ to capture the fact that we use scalar multiplication at the lowest level of the recursion. The solution to this recursion is

$$F(n) = 2n^3 - n^2.$$

Although we have reduced the size of the calculation slightly (by doing fewer additions?) the divide-and-conquer version of matrix multiplication is still an $O(n^3)$ algorithm and even the prefactor remains the same. Therefore we have not made any significant gain for large values of n. Consider, however, the following instead of Eq. (1.3):

$$\begin{aligned} \mathbf{C}_{11} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\ \mathbf{C}_{12} &= \mathbf{M}_3 + \mathbf{M}_5 \\ \mathbf{C}_{21} &= \mathbf{M}_2 + \mathbf{M}_4 \\ \mathbf{C}_{22} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6, \end{aligned} \tag{1.4}$$

where the matrices $\mathbf{M}_1$ to $\mathbf{M}_7$ are defined as

$$\begin{aligned} \mathbf{M}_1 &= (\mathbf{A}_{11} + \mathbf{A}_{22})\,(\mathbf{B}_{11} + \mathbf{B}_{22}) \\ \mathbf{M}_2 &= (\mathbf{A}_{21} + \mathbf{A}_{22})\,\mathbf{B}_{11} \\ \mathbf{M}_3 &= \mathbf{A}_{11}\,(\mathbf{B}_{12} - \mathbf{B}_{22}) \\ \mathbf{M}_4 &= \mathbf{A}_{22}\,(\mathbf{B}_{21} + \mathbf{B}_1) \\ \mathbf{M}_5 &= (\mathbf{A}_{11} + \mathbf{A}_{12})\,\mathbf{B}_{22} \\ \mathbf{M}_6 &= (\mathbf{A}_{21} - \mathbf{A}_{11})\,(\mathbf{B}_{11} + \mathbf{B}_{12}) \\ \mathbf{M}_7 &= (\mathbf{A}_{12} - \mathbf{A}_{22})\,(\mathbf{B}_{21} + \mathbf{B}_{22})\,. \end{aligned} \tag{1.5}$$

These crazy formulae were dreamt up by Strassen [4]. See also [2, chap. 2]. Since the sub-matrices of $\mathbf{A}$ and $\mathbf{B}$ are nowhere commuted in these formulae everything really works and this is a perfectly legitimate way to calculate the product $\mathbf{A}\,\mathbf{B}$. Notice that compared to the 8 multiplications in Eq. (1.3), Eqs. (1.4) and (1.5) involve only 7 multiplications. Admittedly the price to be paid is a far greater number of additions - 18 instead of 4. However the savings from the lower number

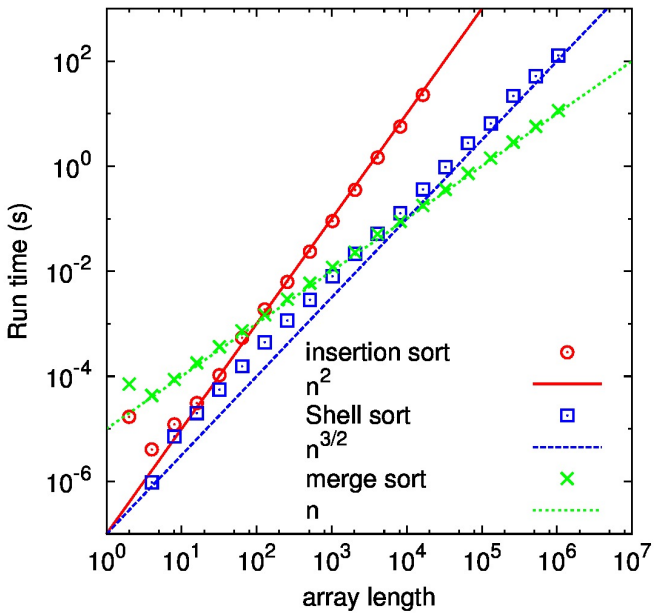


Figure 1.3: Performance measurements of 3 different sorting algorithms: insertion sort (red circles), Shellsort with powers of two increments (blue squares) and mergesort (green crosses). The solid lines are guides show different theoretical scalings.

of multiplications propagates through the recursion in a way which more than compensates for the (subleading) contribution from the increased number of additions. The computational complexity satisfies the recursion

$$F(n) = 7\,F\left(\frac{n}{2}\right) + 18\,\left(\frac{n}{2}\right)^2,$$

with the starting condition $F(1) = 1$. The solution to this recursion is

$$F(n) = n^{\frac{\log(7)}{\log(2)}} - 6n^2. \tag{1.6}$$

Strassen's algorithm is therefore approximately $O(n^{2.81})$! I think this is astounding.

For large problems, considerations of computer performance and algorithmic efficiency become of paramount importance. When the runtime of a scientific computation has reached the limits of what is acceptable, there are two choices:

- get a faster computer (increase the number of FLOPS)
- use a more efficient algorithm

Generally one must trade off between algorithmic efficiency and difficulty of implementation. As a general rule, efficient algorithms are more difficult to implement than brute force ones. This is an example of the "no-free-lunch" theorem, which is a fundamental principle of computational science.

1.3 Sorting

In its simplest form, the sorting problem is the task of designing an algorithm which can transform a list of integers, $\{n_1, n_2, \ldots n_N\}$ into ascending order with the fewest possible number of comparisons. There are many sorting algorithms. We consider a few of them here (see [5] for many others) to illustrate the point that putting some thought into how to solve a problem efficiently can deliver enormous efficiency savings compared to a brute force approach. Figure 1.3 shows compares performance measurements on the three sorting algorithms known as insertion sort, Shellsort and mergesort for a range of list lengths. It is clear that the choice of algorithm has a huge impact on the execution time when the length of the list is large.

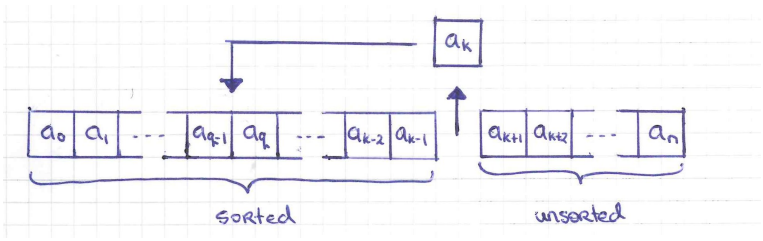
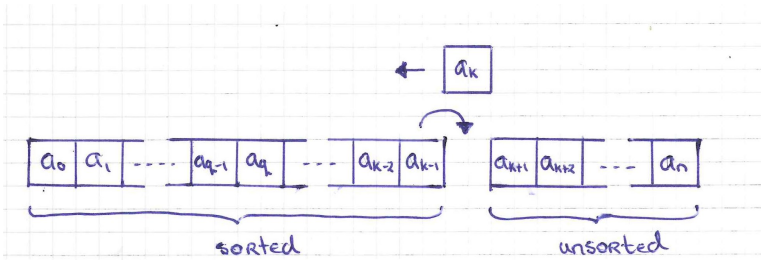


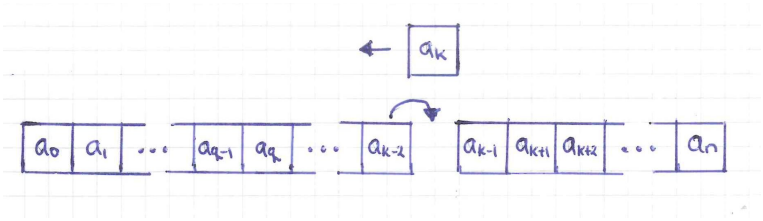
Figure 1.4: General idea of the insertion sort algorithm. At step k of the algorithm, the elements $a_1 \ldots a_{k-1}$ are already sorted. We need to take the next list element a_k, find the value of q in the range $0 \leq q < k-1$ such that $a_{q-1} \leq a_k < a_q$ and insert a_k between a_{q-1} and a_q.

1.3.1 Insertion sort

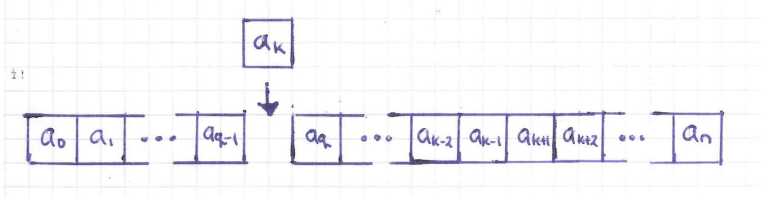
Insertion sort is one of the more "obvious" way of sorting a list of numbers. It is often quoted as the algorithm of choice for card players sorting their hand. It is illustrated in Fig. 1.4. We begin by placing the first two list elements in order. We then take the third list element, step through the first two (which are already in order) and insert it when the correct location is found. We then repeat this process with the fourth, fifth and all remaining elements of the list. At step n of the process, the preceding n-1 elements are already sorted so insertion of element n in the correct place results in a list in which the first n are sorted. This procedure can be implemented "in place" meaning that we do not need to allocate additional memory to sort the array (this is not true of some other sorting procedures). This is achieved as follows. Assuming that the first $k-1$ elements are sorted, we first remove a_k from the array opening up a gap into which other elements can move:



We check if $a_k < a_{k-1}$. If not, a_k is already in the right place and we put it back where it was and proceed to the next step. Otherwise we copy a_{k-1} into the gap and compare a_k to a_{k-2}.



We repeat this process, moving the sorted elements to the right until we either find a_{q-1} such that $a_k > a_{q-1}$ or we reach the beginning of the array. In either case we insert a_k into the gap and proceed to the next step.



The resulting array now has the first k elements sorted.

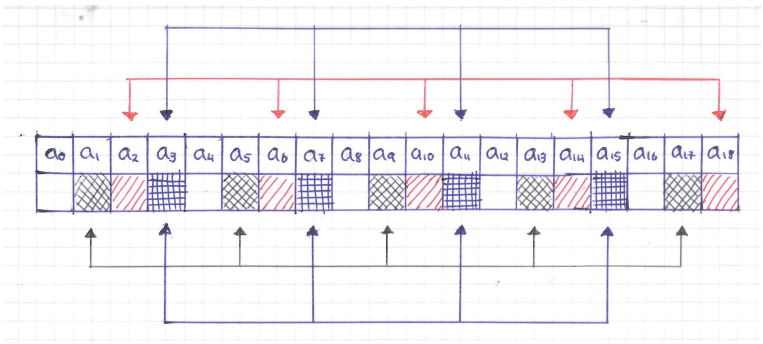
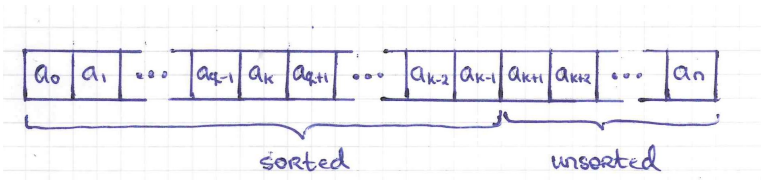


Figure 1.5: Graphical representation of a partial sort of the array $\{a_0, a_1, \ldots a_{18}\}$ with increment 4.



We repeat this process until the entire array is sorted. For each of the n elements of the array we have to make up to $n-1$ comparisons. The complexity of the insertion sort algorithm is therefore $O(n^2)$.

1.3.2 Partial sorting and Shell's method

A partial sort with increment q of a list of numbers is an ordering of the array in which all subsequences with interval q are sorted. The concept is best illustrated with an example. Fig. 1.5 shows a partial sort of the array $\{a_0, a_1, \ldots a_{18}\}$ with increment 4. The array itself is not sorted globally but the subarrays $\{a_0, a_4, a_8, a_{12}, a_{16}\}$, $\{a_1, a_5, a_9, a_{13}, a_{17}\}$, $\{a_2, a_6, a_{10}, a_{14}, a_{18}\}$ and $\{a_3, a_7, a_{11}, a_{15}\}$ are each individually sorted. A small modification to the insertion sort algorithm described above allows one to produce partial sorts: at each step of the algorithm we step through the sorted section of the array in increments of q instead of increments of 1. Obviously a partial sort with increment 1 is a full sort.

While the usefulness of a partial sort may not be immediately obvious, partial sorts can be used to significantly speed up the insertion sort while retaining the "in-place" property. An improvement on the insertion sort, known as ShellSort (after its originator [6]), involves doing a series of partial sorts with intervals, q, taken from a pre-specified list, Q. We start from a large increment and finish with increment 1 which produces a fully sorted list. As as simple example we could take the intervals to be powers of 2: $Q = \left\{2^i \,:\, i = i_{\max}, i_{\max}-1, \ldots, 2, 1, 0\right\}$ where $i_{\max}$ is the largest value of i such that $2^i < n/2$. The performance of the ShellSort algorithm depends strongly on the choice of the sequence Q but it is generally faster on average than insertion sort. This is very counter-intuitive since the sequence of partial sorts always includes a full insertion sort as its final step! The solution to this conundrum is to realise that each partial sort allows elements to move relatively long distances in the original list so that elements can quickly get to approximately the right locations. The initial passes with large increment get rid of large large amounts of disorder quickly, and leaving less work for subsequent passes with shorter increments to do. The net result is less work overall - the final (full) sort ends up sorting an array which is almost sorted so it runs in almost $O(n)$ time.

How much of an overall speed-up can we get? Choosing Q to be powers of 2 for example, scales on average as about $O(n^{3/2})$ (but the worst case is still $O(n^2)$). See Fig. 1.3. Better choices include $Q = \left\{(3^i - 1)/2 \,:\, i = i_{\max}, i_{\max}-1, \ldots, 2, 1, 0\right\}$ for which the average scales as about $O(n^{5/4})$ and the worst case is $O(n^{3/2})$ [5]. The general question of the computational complexity of ShellSort is still an open problem.

1.3.3 Mergesort

A different approach to sorting based on a divide-and-conquer paradigm is the mergesort algorithm which was originally invented by Von Neumann. Mergesort is based on the merging of two arrays

which are already sorted to produce a larger sorted array. Because the input arrays are already sorted, this can be done efficiently using a divide-and-conquer approach. Here is an implementation of such a function in Python:

```
def merge(A,B):
    if len(A) == 0:
        return B
    if len(B) ==0:
        return A
    if A[0] < B[0]:
        return [A[0]] + merge(A[1::],B)
    else:
        return [B[0]] + merge(A,B[1::])
```

This function runs in $O(n)$ time where n is the lenght of the output (merged) array. Can you write down the recursion satisfied by $F(n)$ and show that this function executes in $O(N)$ time? Armed with the ability to merge two sorted arrays together, a dazzlingly elegant recursive function which performs a sort can be written down:

```
def mergeSort(A):
    n=len(A)
    if n == 1:
        return A    # an array of length 1 is already sorted
    else:
        m=n/2
        return merge(mergeSort(A[0:m]), mergeSort(A[m:n]))
```

How fast is this algorithm? It turns out to be $O(n\,\log(n))$ which is a significant improvement. We can see this heuristically as follows. Suppose that n is a power of 2 (this assumption can be relaxed at the expense of introducing additional technical arguments which are not illuminating). Referring to Fig. 1.2, we see that the number of levels, L, in the recursion is $n = 2^L$ from which we see that $L = \log(n)/\log(2)$. At each level, m, in the recursion, we have to solve 2^m sub-problems, each of size $n/2^m$. The total work at each level of the recursion is therefore $O(n)$. Putting these two facts together we can see that the total work for the sort is $O(n\,\log(n))$. The performance of mergesort is shown in Fig. 1.3. It clearly becomes competitive for large n although the values of n reached in my experiment don't seem to be large enough to clearly see the logarithmic correction to linear scaling. Note that mergesort is significantly slower for small arrays due to the additional overheads of doing the recursive calls. There is an important lesson here too: the asyptotically most efficient algorithm is not necessarily the best choice for finite sized problems!

Bibliography

[1] David Goldberg. What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv., 23(1):5–48, 1991. http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html.

[2] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Numerical recipes: The art of scientific computing (3rd ed.). Cambridge University Press, New York, NY, USA, 2007.

[3] Big o notation, 2014. http://en.wikipedia.org/wiki/Big_O_notation.

[4] V. Strassen. Gaussian elimination is not optimal. Numerische Mathematik, 13(4):354–356, 1969.

[5] D. E. Knuth. Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley Professional, Reading, Mass, 2 edition edition, 1998.

[6] D. L. Shell. A high-speed sorting procedure. Commun. ACM, 2(7):30–32, 1959.

Notes 2: Introduction to data structures

2.1 Recursion

2.1.1 Recursive functions

Recursion is a central concept in computation in which the solution of a problem depends on the solution of smaller copies of the same problem. Recursion is a conceptually different approach to thinking about numerical algorithms. It stands in contrast to iteration which is the more familiar way of thinkin about algorithms (at least to most people with backgrounds in mathematics, physics or engineering). All recursive procedures have an alternative formultion as iterative procedures. We will take only a superficial look at recursion in this course since it provides a very neat way to represent certain useful numerical functions and data structures. Computer Science texts go into much greater depth.

Most modern programming languages support recursion by allowing a function to call itself. A recursive function usually consists of one or more recursive cases (values of the inputs for which the function calls itself) and one or more so-called base cases (values of the inputs for which the function returns a results trivially without recurring). An important consideration when designing recursive functions is to make sure that a base case is always reached for allowed values of the input data. Failure to do this can easily result in non-terminating infinite recursions which can be rather difficult to debug. Consider the following classical example which implements the factorial function as a recursive function:

```
int f(int n)
{
  if (n == 0)                // Base case
  { return 1;
  }
  else
  { return n * f(n-1);       // Recursive case
  }
}
```

If you have never encountered the concept of a recursive function before, it takes some thought to see that $f(n) = n!$. One way to understand what is happening is to write out the set of function executions which occur when the function is called. Consider, for example when we calculate $f(4)$. Using brackets to keep track of the levels of the recursion we have

```
f(4) = 4 * f(3)              // Input 3 calls recursive case
     = 4*(3*(f(2))           // Input 2 calls recursive case
     = 4*(3*(2*f(1))         // Input 1 calls recursive case
     = 4*(3*(2*(1*f(0))))    // Input 0 calls base case returning 1
     = 4*(3*(2*(1*1)))       // Multiplications now go from inside the out
     = 4*(3*(2*1))
     = 4*(3*2)
     = 4*6
     = 24
```

We have already seen some less trivial examples of recursive functions in the "divide-and-conquer" algorithms introduced in Secs. 1.2 and 1.3. Recursion is the natural framework for thinking about

divide-and-conquer algorithms. If $F(n)$ represents the number of FLOPs required to solve a problem of size n, we have seen that for divide-and-conquer algorithms, $F(n)$ satisfies a recurrence relation. In the literature on the analysis of the computational complexity of algorithms, the so-called "Master Theorem" provides asymptotic bounds on the complexity of recursively defined divide-and-conquer algorithms. Here we just provide the statement for the purposes of application.

Theorem 2.1.1. *Let a be an integer greater than or equal to 1 and b be an integer greater than 1. Let c be a positive real number and d a nonnegative real number. Given a recurrence of the form*

$$F(n) = \begin{cases} aF(n/b) + n^c & \text{if } n > 1 \\ d & \text{if } n = 1, \end{cases}$$

then if n is a power of b there are three cases:

1. *if $\log_b(a) < c$ then $F(n) = O(n^c)$*
2. *if $\log_b(a) = c$ then $F(n) = O(n^c \log(n))$*
3. *if $\log_b(a) > c$ then $F(n) = O(n^{\log_b(a)})$.*

You should check whether this result applies to the examples with have seen already. For more information on recurrence relations and a proof of the Master Theorem see [1]

2.2 Linked lists

2.2.1 Structural recursion

A data structure is a particular way of organizing data in a computer so that it can be used efficiently. Simple examples of data structures are linear arrays (which allow a list of data to be stored and accessed without assigning a different variable to store each element of the list) or a struct object in C which allows variables or objects of several different types to be grouped together into a single structure which can be easily manipulated inside a code. Some more sophisticated and powerful data structures emerge when the concept of recursion, which up until now we have thought of as a property of a function, is applied to a data type. The resulting recursive data types are data structures which are defined in terms of themselves. This is sometimes referred to as "structural recursion". Examples include structures like linked lists, stacks, queue and trees which facilitate very efficient implementation of certain numerical procedures.

2.2.2 Pointers

Before studying recursive data structures it is helpful to understand the concept of a pointer. Pointers are one of the most powerful features of modern programming languages like C/C++ or Fortran90. They are also one of the easiest features to use incorrectly, often resulting in bugs which are highly nontrivial to find. As a result of the potential for accidental (or deliberate) mis-use, some programming languages (such as Java and MatLab) don't support pointers natively although they provide complex data types which implement pointer-like functionality with additional checking to prevent mis-use.

In simple terms, a pointer is a type of variable which stores the address of ("points to") a regular variable. See Fig. 2.1. A pointer references a location in memory. In order to be useful, it is usually necessary to read the value of the variable which a pointer points to. The operation of obtaining the value of a variable referenced by a pointer is known as dereferencing the pointer. Most of the applications of pointers exploit the fact that it is often much cheaper in time and space to copy and dereference pointers than it is to copy and access the data to which the pointers point.

In C/C++ the syntax for defining and dereferencing a pointer involves the symbol *. The syntax for obtaining the address of a variable to store in a pointer involves the symbol &. Here is an example:

```
int a = 6;              // Define an int a and set its value to be 6
int *ptr = NULL;        // Define a pointer ptr and set its value to be NULL

ptr = &a;               // Set ptr to point to a
cout << *ptr << endl; // Print the value of the variable pointed to by ptr
```

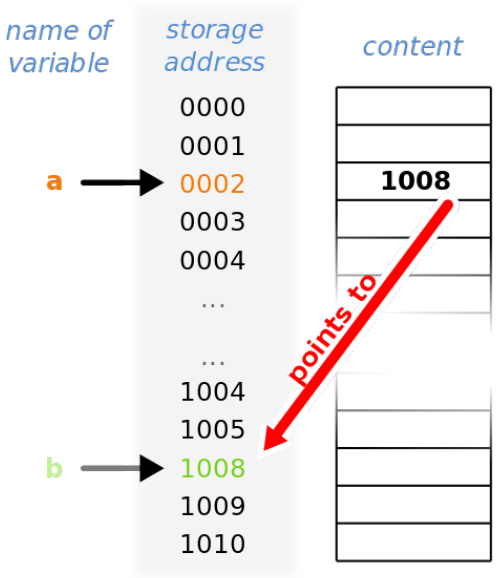


Figure 2.1: (from wikipedia) Graphical illustration of pointers. The value stored in the pointer shown is the address of the variable b.

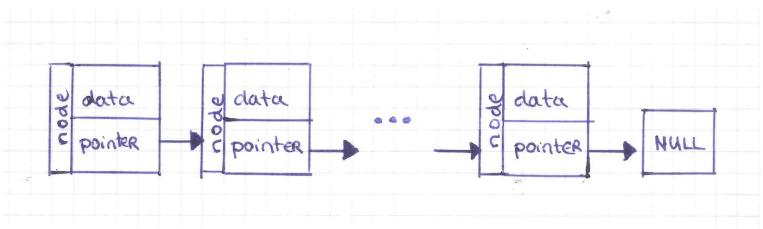
The value NULL has a special meaning in the C/C++ language. A NULL pointer has a value reserved for indicating that the pointer does not refer to a valid object.

The power and dangers of using pointers are associated with the fact that they can be directly manipulated as numbers. As a result they can be made to point to unused addresses or to data which is being used for other purposes. This can lead to all sorts of programming errors. A pointer which does not have any address assigned to it is called a "wild" pointer. Any attempt to use such uninitialized pointers can cause unexpected behavior, either because the initial value is not a valid address, or because using it may damage other parts of the program. The result is often a segmentation fault or other storage violation. Even worse, in systems with explicit memory allocation (in C this is implemented using the malloc() function and its variants), it is possible to create a "dangling" pointer by accidentally deallocating the memory region it points into. This type of pointer is dangerous and subtle because a deallocated memory region may contain the same data as it did before it was deallocated but may be then reallocated and overwritten. This can lead to the most evil types of "random" bugs in which a program sometimes runs correctly (if the memory pointed to by the dangling pointer happens not to be overwritten before it is used) but sometimes runs incorrectly or crashes with a segmentation fault (if the memory pointer to by the dangling pointer is reallocated for some other purpose before it is used).

Let us now look at how pointers can be used to do useful things.

2.2.3 Linked lists

A linked list is a data structure which contains a contains a sequence of nodes. In the simplest type of linked list (the so-called singly linked list), each node consists of some data and a pointer to the next node in the list. Usually the last node in the list has a NULL pointer indicating the end of the list:



In some respects a linked lists is like a conventional linear array in the sense that is stores a

sequence of data objects. It has some advantages over a conventional linear array however:

- Linked lists are dynamic data structures. The memory is allocated as the items are added to the list so it is not necessary to know the length of the list in advance.
- Elements can be efficiently inserted or removed from the list without reallocation or reorganization of the rest of the elements. This is because the list elements do not need to be stored contiguously in memory.
- They form the building blocks of more complicated data structures like trees and networks which are not easily implemented as linear arrays.

There are some disadvantages too:

- The biggest disadvantage is that linked lists only provide sequential access to list elements. To get to node i requires traversing the list from the beginning. This makes direct random access to list elements very inefficient.
- Because each node stores a pointer in addition to an item of data, a linked list takes up more memory than a linear array to store an equivalent amount of data.

Since each node of the list points to another node which points to another node etc, it is natural to define a linked list as a recursive data structure: a linked list is either empty, or it is a node which points to a linked list. Here is a simple C++ class which implements such a definition for a linked list which stores integer data:

```
class List
{
   int n;                          // The data is just an int
   List *next;                     // Pointer to the next node in the list
};
```

We can add to this class a method which, given a pointer to a list and an integer, inserts a new item at the head of the list and then returns a pointer to the modified list:

```
List * List::insertHead(List *list, int N)
{
   List *new_node = new List();     // Create a new node to be the new head
   new_node->n = N;                 // Assign the data to the new node
   if(list != NULL)
   {  new_node->next = list;        // New node points to the previous list
   }
   return new_node;                 // Return a pointer to the new list
}
```

If you can understand this piece of code then you really understand pointers. Let us now add a method which, given a pointer to a list and an integer, inserts a new item at the tail of the list and then returns a pointer to the modified list. One way to do this is via recursion:

```
List * List::insertTail(List *list, int N)
{
   if(list == NULL)                 // Base case
   {
      List *new_node = new List();  // Create a new node
      new_node->n =N;               // Store the data
      return new_node;              // Return the new node
   }
   else                             // Recursive case
   {                                // Recurse on list->next
     list->next = insertTail(list->next, N);
     return list;
   }
}
```

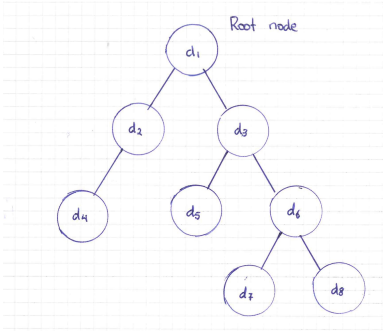
If you can understand this piece of code then you really understand recursive functions. Does this code really insert a new node at the tail of the list? Consider the following argument:

- If list has length 0, the base case holds. The method returns to a list consisting of only a new node containing the integer N. This is the correct answer.
- If list has length $n > 0$ then the recursive call is applied to the List list.next which has length $n - 1$.
- Let us assume that this recursive call gives the correct result for a list of length $n - 1$. That is to say we assume the recursive call returns a pointer to the List list.next (containing all nodes after the first) with a new node containing the integer N inserted at its rear. Given this assumption, storing a pointer to this List into list->next and returning list produces the correct result for the list of length n.
- By induction, we therefore obtain the correct result for a List of any finite length.

Many operations on linked lists are most easily written using recursive methods. This is natural, because linked lists are themselves defined recursively. A basic implementation of the above class can be downloaded from the class website for you to play around with.

2.3 Binary trees

It is fairly easy to implement linked lists in alternative ways to the approach described in Sec. 2.2 which avoid recursion. Some more complicated data structures however, are very difficuilt to implement without recursion. One of the simplest and most useful of such structures is a binary tree. A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child. The top node in the tree is called the root node. As with linked lists, a binary tree data structure usually contains some data at each node in addition to pointers to the left and right child nodes. Here is a sketch of a small tree with 8 nodes storing the list of data items $\{d_1, d_2, \ldots d_8\}$ in "breadth-first" search order :



Such a structure is defined recursively as follows:

```
class BinaryTree
{
  Datum data;
  BinaryTree *L;
  BinaryTree *R;
};
```

Here the Datum object represents a custom defined container object which can contain whatever we want.

2.3.1 Binary tree search

Binary trees have useful applications in search and sorting problems. One particularly important application of trees in search is a structure called a binary search tree. Consider an array of N key-value pairs, $\{(k_1, d_1), (k_2, d_2), \ldots (k_N, d_N)\}$ which can be sorted on the key values. This is to say, the list can be arranged so that $k_1 \leq k_2 \leq \ldots \leq k_N$. For a random key value, k, between k_1 and k_N we wish to return the data value associated with the key k. The naive approach would be to simply iterate through the sorted list one by one, comparing the key of each list element with k until we either

find one that matches or reach a value which exceeds k (in which case the key k is not in the list). On average this is clearly an $O(N)$ operation. By storing the list in a binary search tree, this operation can be accomplished in $O(\log(N))$ time. A binary search tree, sometimes also called an ordered or sorted binary tree, is a binary tree with a key-value pair at each node withoutthe special property that the key of any node is larger than the keys of all nodes in that node's left child and smaller than the keys of all nodes in that node's right child. A small modification of the above incorporates the search key (we take the key to be an integer) :

```
class BinaryTree
{
  int key;
  Datum data;
  BinaryTree *L;
  BinaryTree *R;
};
```

Fast search is achieved by descending through the tree from root to leaves. At each level, a single comparison with the node key determines whether to go left or right. The total number of comparisons required to reach the leaf level is obviously equal to the number of levels which (assuming that the tree is "balanced") scales as $\log(N)$. We begin by examining the root node. If the tree is null, the key we are searching for does not exist in the tree. Otherwise, if the key equals that of the root, the search is successful and we return the value at that node. If the key is less than that of the root, we search the left subtree. Similarly, if the key is greater than that of the root, we search the right subtree. This process is repeated until the key is found or the remaining subtree is null. If the searched key is not found before a null subtree is reached, then the item must not be present in the tree. This is easily expressed as a recursive algorithm which returns a pointer to the node containing the matching key (or the NULL pointer in the case that the key is not matched):

```
BinaryTree * search(BinaryTree *T, int k)
{
  if (T == NULL)
  {   return NULL;  // base case
  }
  else if (T->key == k)
  {   return T;
  }
  else if (T->key < k)
  {    return search(T->L, k);
  }
  else if (T->key > k)
  {    return (T->R, k);
  }
}
```

Let us use Theorem 2.1.1 to verify our expectation that binary tree search should be an $(O(\log(N))$ operation. Assuming that the tree is balanced, both the left and right child trees will have approximately half the number of nodes. The amount of work which we have to do at each iteration is therefore (at most) 4 comparisons followed by a search on a tree of half the size. Mathematically

$$F(n) = F\left(\frac{n}{2}\right) + 4,$$

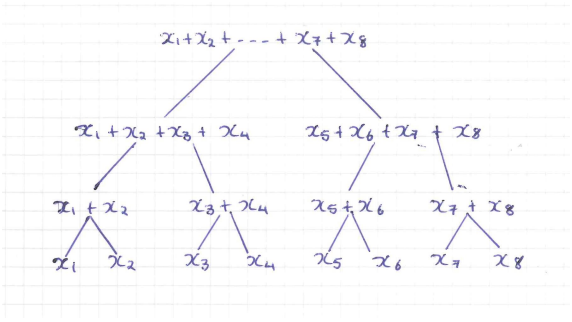
giving $a = 1$, $b = 2$ and $c = 0$ in the notation of Theorem 2.1.1. We see that $\log_b(a) = \log_2(1) = 0 = c$. We are therefore in case 2 and $F(n) = O(\log(n))$.

2.3.2 Searching lists of partial sums - Fenwick trees

The following problem arises so frequently in simulation of stochastic processes (see [2]) that it is worth devoting a little time to study how to solve it efficiently. We are given a list of pairs, $\{(x_1, k_1), (x_2, k_2), \ldots (x_N, k_N)\}$, in which each x_i is a real number and each k_i is an integer. In the application to stochastic simulation, we have a list of possible events which can occur in the system labelled $k_1, \ldots, k_N$ and an associated list of rates $x_1, \ldots, x_N$, at which these events occur. From the list of rates we can calculate the list of partial sums,

$$\{s_i : i = 0, 1 \ldots N\} \qquad \text{where } s_i = \sum_{j=1}^{i} x_j,$$

(we take $s_0 = 0$). The computational challenge is the following: given a number $x \in (0, s_N]$, can we efficiently find i such that $s_{i1} < x \leq s_i$. Essenatially we want to know in which interval of the list of partial sums does the value x lie. In applications to stochastic simulation we usually want to return the index k_i since this determines which event happend next. Obviously we can solve this problem by linear search on the sorted list of partial sums. One average this would require $O(N)$ comparisions. In fact we can use the ideas of Sec.2.3.1 to design a data structure which can solve this problem in $O(\log(N))$ time. The required data structure is a binary tree in which each node stores the sum of the x values of all the nodes in its left and right children. Here is what it looks like for the sequence of values $x_1, x_2, \ldots x_8$:



Such a structure is called a Fenwick tree. If we leave aside the question of how to build the tree starting from the initial list, the process of searching for the interval containing a particular value x is rather similar to the binary tree search. We compare x to the sum of the values stored in left child and if x is less than this value then we search for x on the left branch. If not, then we subtract from x the value of all the nodes on the left branch and search for the result on the right branch. The subtraction is the bit which is easy to get wrong here. Check the diagram above to convince yourself that this is correct. When we get to a leaf node then we return the data stored there. Here is a recursive implementation of this process in C++

```
Datum BinaryTree::search(BinaryTree *T, double x)
{
  if(T->L==NULL && T->R==NULL)
  { // Base case: we have reached a leaf node. Return the data stored there
    return T->data;
  }
  else
  {
    // Recursive case: decide whether to go left or right.
    double y=(T->L->data).x;  // Get the sum of the left child
    if(x <= y)
    { // Search left child
      return search(T->L, x);
    }
    else
```

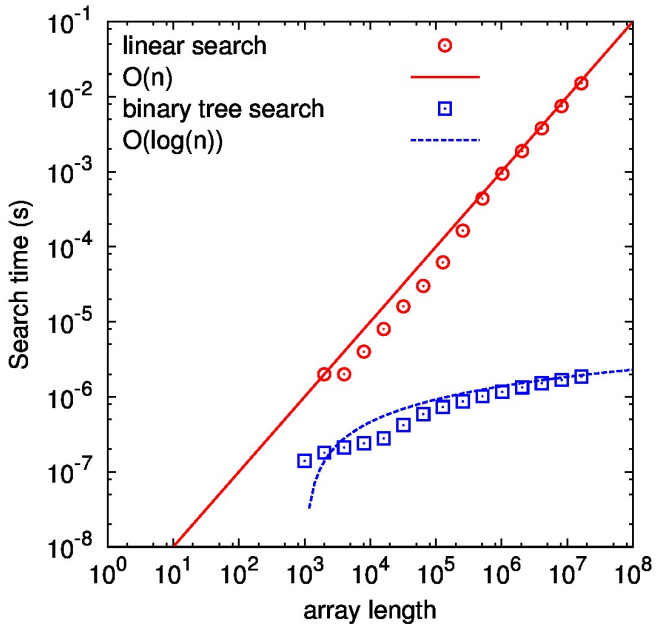


Figure 2.2: Comparison of the performance of binary tree search (blue squares) and linear search (red circles) for the problem of searching the running total of an array of real numbers.

```
    { // Search right child
      return search(T->R, x-y);
    }
  }
}
```

When doing large scale stochastic simulations, the extra work associated with implementing such a data structure is well worth the increase in speed of the resulting code. Fig. 2.2 shows the results of doing some performance measurements of search on a Fenwick tree compared to naive linear search. If you have not used binary search algorithms in your programming before, the increase in speed for large arrays is truly awesome to behold! An implementation of a Fenwick tree is available from the class website for you to play around with. In particular, this implementation contains a recursive function which builds the tree in the first place. Building the tree is an $O(N)$ operation. Another important point about Fenwick trees and binary search trees in general which makes them practical for efficient implementation of Monte Carlo simulations is that once a tree has been built, the operations of insertion, deletion and updating of nodes can also be done in $O(\log(N))$ time. This means that it is not necessary to rebuild the tree each time the underlying array of rates changes. A long sequence of insertions and deletions can however result in the tree becoming unbalanced resulting in a loss of the fast search property as the number of levels grows (the extreme case of a completely unbalanced tree has N levels and search becomes equivalent to linear search). In practice, the tree may need to be rebalanced at occasional intervals. These are more specialise topics which we will not cover in this course but it is good to be aware of them if you end up using these methods in your research.

Bibliography

[1] K. Bogart and C. Stein. CS 21/math 19 - course notes. `https://math.dartmouth.edu/archive/m19w03/public_html/book.html`, 2003.

[2] J. L. Blue, I. Beichl, and F. Sullivan. Faster monte carlo simulations. Phys. Rev. E, 51(2):R867–R868, 1995.

Notes 3: Interpolation, numerical derivatives and numerical integration

3.1 Interpolation

3.1.1 Lagrange polynomials

Suppose we are given $n+1$ points $\{x_0, x_1, \ldots x_n\}$ in an interval $[a_1, a_2]$ and $n+1$ associated values, $\{y_0, y_1, \ldots y_n\}$, which we assume are samples of the value of an unknown function, $f(x)$, at these values. A common computational task is to construct a function $g(x)$ which approximates $f(x)$ at any value of x in the interval $[a_1, a_2]$. This is known as interpolation. Of course, there is no way to solve the task as stated: an infinite amount of information can be encoded in $f(x)$ which cannot be recovered from a finite number, $n+1$, of function values. A reasonable candidate for $g(x)$ is the polynomial of degree N which passes through all $n+1$ known points $\{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\}$. Such a polynomial exists and is unique. It it known as the Lagrange polynomial:

$$P(x) = \sum_{i=0}^{n} y_i \, p_i(x) \tag{3.1}$$

where

$$p_i(x) = \prod_{\substack{j=0 \\ i \neq j}}^{n} \frac{x - x_j}{x_i - x_j}.$$

You should check that this formula does as it says. Note that it includes the special cases of linear interpolation ($n = 1$) which will be familiar to everyone. Fig. 3.1 shows the Lagrange polynomial approximating the Gaussian function,

$$f(x) = e^{-\frac{x^2}{2}}, \tag{3.2}$$

using $n = 5, 7, 9$ and 11 equally spaced samples in the interval $[-3, 3]$. Obviously, every interpolating polynomial can also be used for extrapolation of $f(x)$ outside of the sample interval. Fig. 3.1 illustrates that this is (usually) not a terribly good idea.

3.1.2 Neville's algorithm

The question of how to compute Eq. (3.1) efficiently is a non-trivial one. From a numerical point of view, it is much more common to compute the value of the interpolating polynomial at the required point x directly than to compute the coefficients and then evaluate the resulting polynomial at x. There is some detailed discussion in [1, chap.3] about why this is so. The best algorithm for computing $P_n(x)$ for any value x is known as Neville's algorithm. It is a recursive procedure which builds the required polynomial order by order from intermediate polynomials of lower order, starting from constants (zeroth order polynomials!). We use the representation of Neville's algorithm shown in [1, chap. 3]. If you find this difficult to understand, it might be worth reading the Wikipedia article on Neville's algorithm which shows a slightly different representation [2] in which the intermediate polynomials are indexed differently.

The idea is to construct the tree shown in Fig. 3.2 from the $n+1$ points $\{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\}$. The tree has $n+1$ levels, labeled from 0 to n starting from the right. At level i in the tree there are $n+1-i$ nodes. Let us index the nodes by (i, j) where i is the level in the tree and j is the number of the node within that level. Each node, (i, j) is associated with a polynomial, $P_{j,j+1\ldots j+i}(x)$. The algorithm is specified by the following inductive argument:

- Let us assume that $P_{j,j+1\ldots j+i}(x)$ is the unique polynomial of degree i which passes through the points $\{(x_j, y_j),\ (x_{j+1},\ y_{j+1}), \ldots, (x_{j+i}, y_{j+i})\}$.

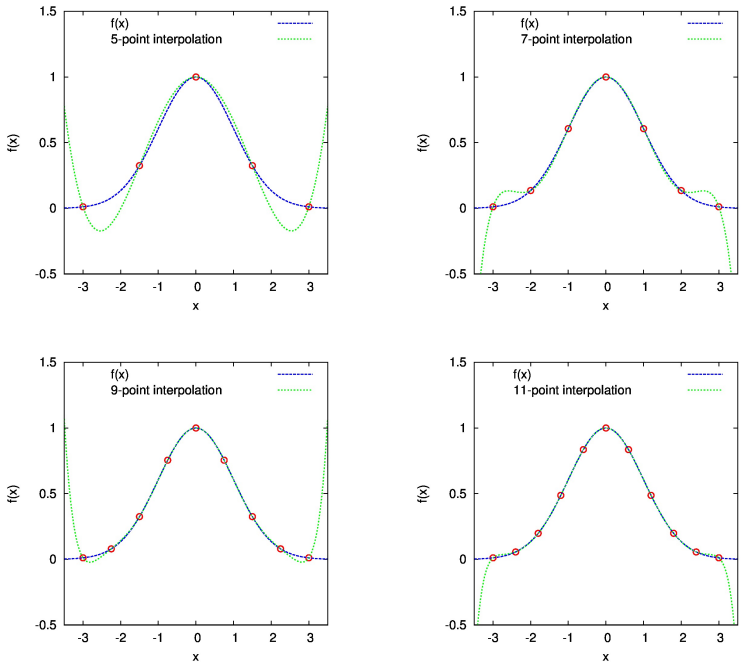


Figure 3.1: Lagrange interpolation of the Gaussian function, Eq. (3.2), with equally spaced sample points in the interval

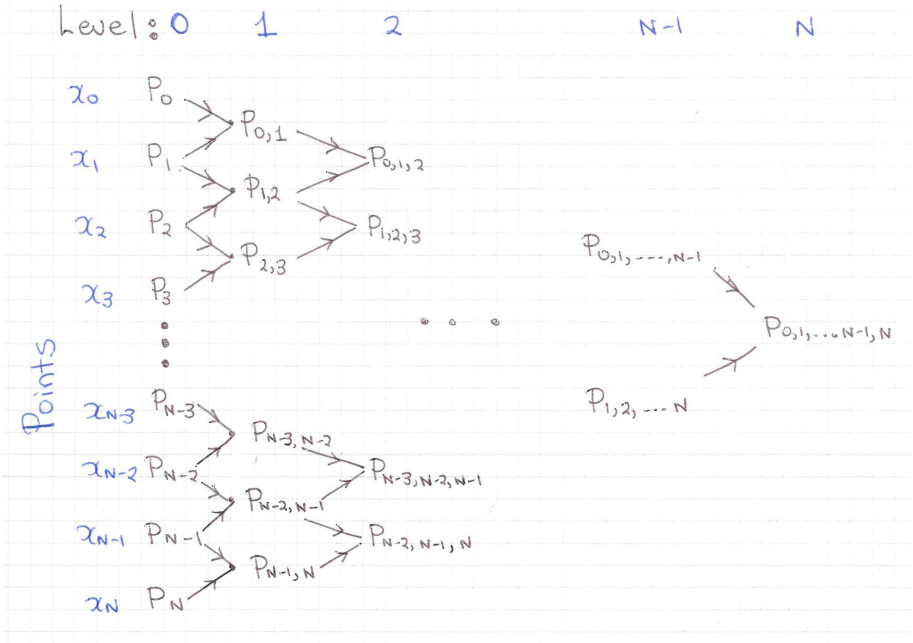


Figure 3.2: Graphical representation of the recursive structure of Neville's algorithm.

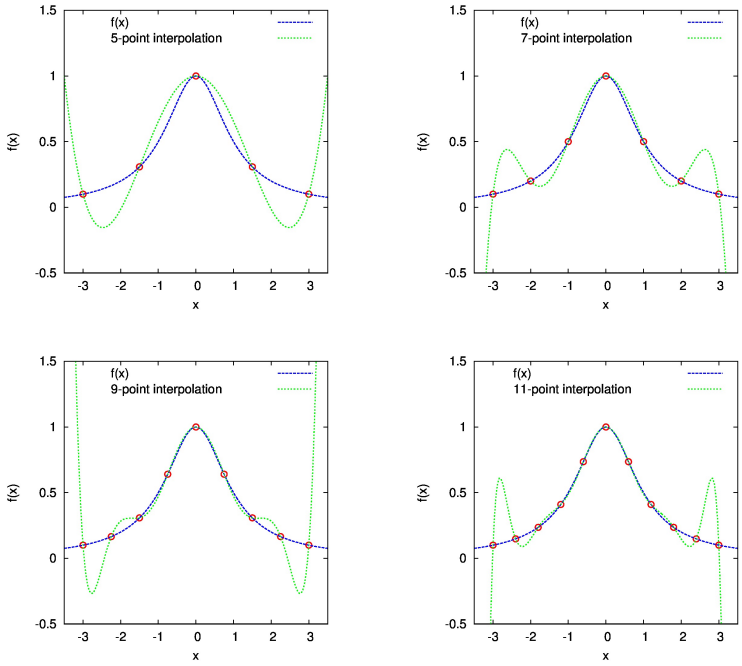


Figure 3.3: Lagrange interpolation of the Runge function, Eq. (3.5), with equally spaced sample points in the interval $[-3, 3]$.

- With this assumption, the polynomial

$$P_{j,j+1\ldots j+i+1}(x) = \frac{(x - x_{j+i+1})\, P_{j,j+1\ldots j+i}(x) - (x - x_j)\, P_{j+1,j+2\ldots j+i+1}(x)}{x_j - x_{j+i+1}} \tag{3.3}$$

  is the unique polynomial of degree $i+1$ which goes through the points $\{(x_j, y_j), (x_{j+1}, y_{j+1}), \ldots, (x_{j+i+1}, y_{j+i}$ Check it to make sure it works!
- The recursion relation (3.3) gives us a way to step from level i to level $i + 1$ in Fig. 3.2. All that remains is to provide the starting condition at level 0:

$$P_i(x) = y_i. \tag{3.4}$$

  This is clearly the unique polynomial of degree 0 passing through the point (x_i, y_i). By induction, the polynomial $P_{0,1\ldots N}(x)$ at level N is the required Lagrange polynomial.

3.1.3 The concept of conditioning

The condition number of a function with respect to an argument measures how much the output value of the function can change for a small change in the input argument. It is used to measure how sensitive a function is to changes or errors in the input. This concept is often applied in a context where the "function" is the solution of a numerical problem and the "argument" is the data going into the problem. In the present context, the problem is the calculation of the interpolating polynomial and the argument is the set of sample points. A problem with a low condition number is said to be well-conditioned, while a problem with a high condition number is said to be ill-conditioned. It is important to realise that the condition number is a property of the problem itself, not of the algorithm used to solve it or the precision with which this algorithm is implemented. For ill-conditioned problems, however, roundoff error coming from finite precision can act as a source of error in the input which can lead to large and unpredictable variations in the output.

In the context of interpolation, we expect intuitively that as the number of samples increases (assuming that we keep the interval $[a_1, a_2]$ fixed) then the interpolating polynomial, $P(x)$, will converge to $f(x)$. Although this seemed to be the case with the example shown in Fig. 3.1, such convergence

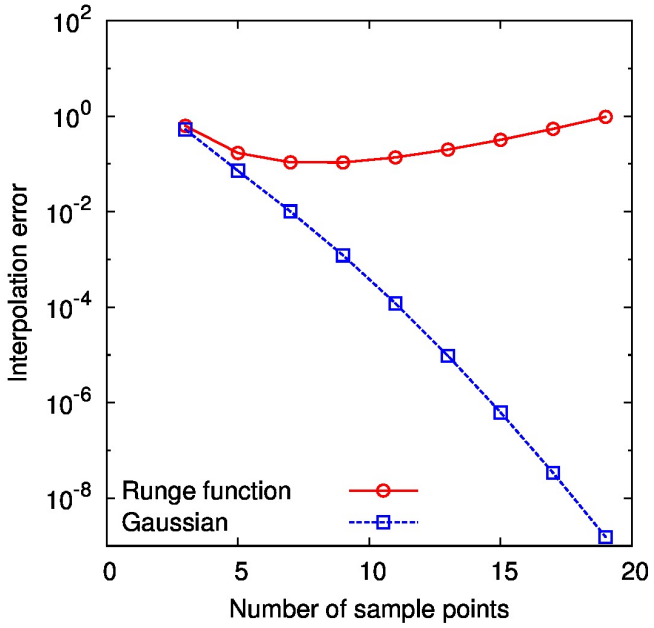


Figure 3.4: Interpolation error as a function of number of sample points for Lagrange interpolation of the Runge function, Eq. (3.5), and Gaussian function, Eq. (3.2), with equally spaced sample points in the interval $[-3, 3]$. For the Gaussian function the interpolation error decreases monotonically with the order and the interpolating polynomial converges to the treu function. For the Runge function, the error goes down at first but then starts to grow again!

is in fact not assured. Consider the so-called Runge function [3]

$$f(x) = \frac{1}{1+x^2}, \tag{3.5}$$

which is not all that different from the Gaussian example used above. Fig. 3.3 shows the Lagrange polynomial approximating this function using $n = 5, 7, 9$ and 11 equally spaced samples in the same interval $[-3, 3]$. Despite the fact that Eq. (3.5) is not qualitatively very different from Eq. (3.2), $f(x)$ becomes increasingly badly represented near the edges of the interval as the number of sample points, n, increases. The oscillations between the sample points actually get larger as n increases. The interpolation error can be quantified by calculating

$$E_n = \sqrt{\int_{-a}^{a} |P_n(x) - f(x)|^2 \, dx}. \tag{3.6}$$

This is plotted as a function of n for the Gaussian and Runge examples in Fig. 3.4. The fact that larger n does not necessarily imply smaller error was an important discovery in the history of numerical analysis attributed to Runge. The oscillations of the interpolating polynomial responsible for this effect are known as the Runge Phenomenon. These oscillations are not due to numerical instability but are reflecting the fact that the problem of Lagrange interpolation with equally spaced sample points is ill-posed in the sense that small variations in the input parameters (the sample points) produce large variations in the resulting interpolating polynomial. The problem of Lagrange interpolation is an ill-conditioned problem. From a numerical perspective Lagrange interpolation at high orders is a dangerous business since there is the omnipresent danger that small numerical errors in the sample values will be amplified by this intrinsic instability. This effect is closely related to the problem of over-fitting which you encountered in a different guise in MA930. There is a large literature on interpolation methods which go beyond Lagrange in an attempt to get around these difficulties.

3.2 Numerical differentiation

We now turn our attention to another common computational task: the evaluation of the derivative of a function, $f(x)$, at a point, x.

3.2.1 Finite difference approximations of derivatives

Suppose we have a list of the values, $\{f_i : i = 1 \dots N\}$ of a function, $f(x)$, at a set of discrete points, $\{x_i\}$. Let us further assume, for simplicity, that these points are equally separated with interval h: $x_i = x_0 + ih$. Intuitively, derivatives of f a point can be approximated by differences of the f_i at adjacent points:

$$\frac{df}{dt}(x_i) \approx \frac{f_{i+1} - f_i}{h}.$$

This intuition is made precise using Taylor's Theorem:

Theorem 3.2.1. *If $f(x)$ is a real-valued function which is differentiable $n+1$ times on the interval $[x, x+h]$ then there exists a point, ξ, in $[x, x+h]$ such that*

$$\begin{aligned} f(x+h) &= f(x) + \frac{1}{1!}\, h\, \frac{df}{dx}(x) + \frac{1}{2!}\, h^2\, \frac{d^2 f}{dx^2}(x) + \dots \\ &\quad + \frac{1}{n!}\, h^n\, \frac{d^n f}{dx^n}(x) + h^{n+1} R_{n+1}(\xi) \end{aligned} \tag{3.7}$$

where

$$R_{n+1}(\xi) = \frac{1}{(n+1)!}\, \frac{d^{n+1} f}{dx^{n+1}}(\xi).$$

Note that the theorem does not tell us the value of ξ. We will use Theorem (3.2.1) a lot. Adopting "primed" notation to denote derivatives, Taylor's Theorem to first order ($n = 1$) tells us that

$$f(x+h) = f(x) + h\, f'(x) + O(h^2).$$

In the light of the above discussion of approximating derivatives using differences, we can take $x = x_i$ and re-arrange this to give

$$f'(x_i) = \frac{df}{dx}(x_i) = \frac{f_{i+1} - f_i}{h} + O(h). \tag{3.8}$$

This is called the first order forward difference approximation to the derivative of v. Taylor's theorem makes the sense of this approximation precise: we make an error proportional to h (hence the terminology "first order"). Thus it is ok when h is "small".

We could equally well use the Taylor expansion of $f(x-h)$:

$$f(x-h) = f(x) - h\, f'(x) + O(h^2)$$

Rearranging this gives what is called the first order backwards difference approximation:

$$f'(x_i) = \frac{f_i - f_{i-1}}{h} + O(h). \tag{3.9}$$

It also obviously makes an error of order h. To get a first order approximation of the derivative requires that we know the values of f at two points. The key idea of finite difference methods is by using more points we can get better approximations. Better means the error made in the approximation is proportional to a higher power of h. The general method is to consider a linear combination of the values of the function at a set of points surrounding the point x at which we want to approximate the derivative. We use Taylor's theorem to express everything in terms of the value of the function and its derivatives at the point x and then choose the coefficients in the linear combination to cancel as many powers of h as possible. This is best illustrated by an example. Suppose we wish to derive a finite difference approximation for the derivative of $f(x)$ at x_i using the values of $f(x)$ at the three points $x_{i-1} = x_i - h$, x_i and $x_{i+1} = x_i + h$. We will need the Taylor expansions at x_{i-1} and x_{i+1}:

$$\begin{aligned} f_{i+1} &= f_i + h\, f'(x_i) + \frac{1}{2}\, h^2\, f''(x_i) + O(h^3) \\ f_{i-1} &= f_i - h\, f'(x_i) + \frac{1}{2}\, h^2\, f''(x_i) + O(h^3). \end{aligned}$$

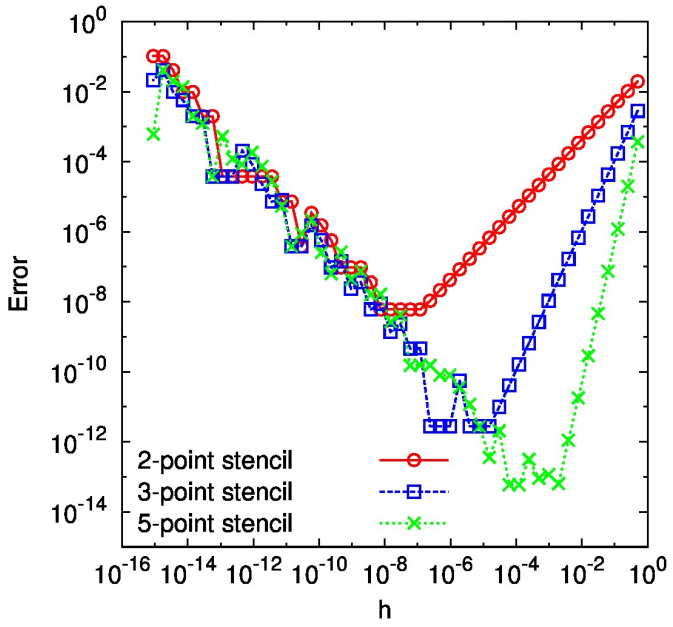


Figure 3.5: Finite difference approximation to the derivative of $f(x) = \sqrt{x}$ at $x = 2$. The plot shows the error as a function of h for several different finite difference formulae.

We now consider the linear combination $a_1\, f_{i-1} + a_2\, f_i + a_3\, f_{i+1}$ and use these Taylor expansions. We obtain

$$a_1\, f_{i-1} + a_2\, f_i + a_3\, f_{i+1} = (a_1 + a_2 + a_3)\, f_i + (a_3 - a_1)\, h\, f'(x_i) + \frac{1}{2}\,(a_3 + a_1)\, h^2\, f''(x_i) + O(h^3).$$

We now choose the values a_1, a_2 and a_3 to satisfy the equations

$$\begin{aligned} a_1 + a_2 + a_3 &= 0 \\ a_3 - a_1 &= 1 \\ a_3 + a_1 &= 0. \end{aligned}$$

This cancels the terms proportional to h^0 and h^2. We obtain $a_1 = -\frac{1}{2}$, $a_2 = 0$ and $a_3 = \frac{1}{2}$. Solving for $f'(x_i)$ then gives

$$f'(x_i) = \frac{f_{i+1} - f_{i-1}}{2\,h} + O(h^2). \tag{3.10}$$

This is called the centred difference approximation. This procedure generalises easily to unequally spaced points although the formulae are not so neat. An astute reader will have noticed that we could have chosen a_1, a_2 and a_3 so as to cancel the terms proportional to h^0 and h^1 and obtained an approximation for $f''(x_i)$. This is indeed how finite difference formulae for higher order derivatives are obtained.

The set of points underpinning a finite-difference approximation is known as the "stencil". Notice that by using a 3-point stencil, we obtained an approximation which is second order accurate in h. A straightforward extension to stencils with more points allow higher order approximations to be derived. For example, a 5-point stencil leads to a 4th order accurate finite difference formula:

$$f'(x_i) = \frac{-f_{i+2} + 8f_{i+1} - 8f_{i-1} + f_{i-2}}{12\,h} + O(h^4). \tag{3.11}$$

From this discussion, it may seem optimal to choose h as small as possible in order to reduce the error. Due to roundoff error, the situation is not so simple when finite difference methods are implemented in finite precision arithmetic. In fact it can be quite difficult to determine a-priori what is

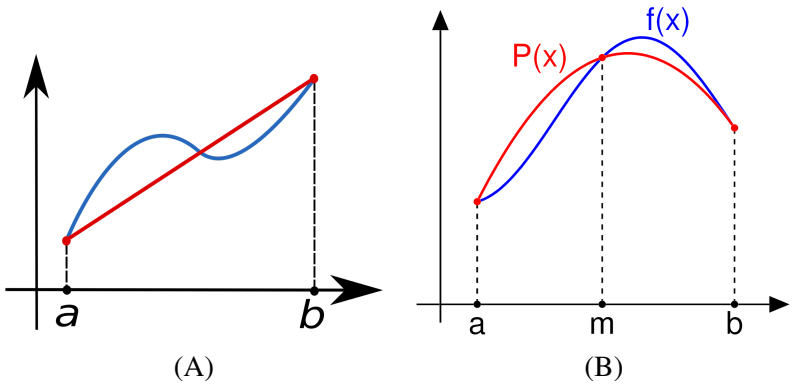


Figure 3.6: (from Wikipedia) Graphical illustration of (A) the Trapezoidal Rule and (B) Simpson's Rule.

the optimal value of h. See Fig. 3.5 to see an illustration of this. With higher order finite difference formulae, h does not have to be very small (around $h = 0.001$ for Eq. (3.11)) before we get to a point where decreasing h further starts to make the estimated value of the derivative worse rather than better!

3.3 Numerical integration

We now turn to the other basic operation of elementary calculus: integration. A common computational task is to estimate the definite integral of some function, $f(x)$ over an interval $[a, b]$:

$$I = \int_a^b f(x)a, dx.$$

3.3.1 Trapezoidal Rule and Simpson's Rule

The classical numerical integration rules construct an approximation from the values of $f(x)$ sampled at $m + 1$ points, $x_0, \ldots x_m$, in the interval $[a, b]$. We think of the interval as being fixed: $x_0 = a$ and $x_n = b$. Thus as m increases, the points get closer together. We further assume that the points are equally spaced with separation $h = (b - a)/n$. The trapezoidal rule, shown in Fig 3.6(A), uses two values ($m = 1$), f_0 and f_1 :

$$I = \frac{h}{2}[f_0 + f_1] + O(h^3). \tag{3.12}$$

Simpson's rule, shown in Fig 3.6(B), uses three values ($m = 2$), f_0, f_1 and f_2:

$$I = \frac{h}{3}[f_0 + 4f_1 + f_2] + O(h^5). \tag{3.13}$$

These formulae and higher order ones (involving the use of more points inside the interval $[a, b]$) can be derived by using Eq. (3.1) to interpolate the function $f(x)$ in the interval $[a, b]$ and then use the fact that polynomials are easily integrated analytically. The determination of the error estimate is less straightforward and is detailed (for those interested) for the trapezoidal rule in [4].

In practice, the region of integration consists of more than a single interval. We can divide a larger interval into n non-overlapping subintervals each of length $h = (b-a)/n$. Eq.(3.12) or (3.13) can then be applied to each subinterval and the result summed to estimate the integral over the entire interval. It is instructive to write out the formulae explicitly to help see the general pattern. Applying Eq.(3.12) to the pair of intervals (x_0, x_1), (x_1, x_2) gives

$$I = h\left[\frac{f_0}{2} + f_1 + \frac{f_2}{2}\right] + 2\,O(h)^3$$

which generalises with n intervals to the so-called "extended" trapezoidal rule:

$$I = h\left[\frac{f_0}{2} + \sum_{i=1}^{n-1} f_i + \frac{f_n}{2}\right] + O\left(\frac{(b-a)^3}{n^2}\right). \tag{3.14}$$

The error term comes from making n errors of $O(h^3)$ (see Eq. (3.12)) with $h = (b-a)/n$. Similarly, applying Eq. (3.13) to the pair of pairs of intervals (x_0, x_1), (x_1, x_2) and (x_2, x_3), (x_3, x_4) gives

$$I = h\left[\frac{f_0}{3} + \frac{4\,f_1}{3} + \frac{2\,f_2}{3} + \frac{4\,f_3}{3} + \frac{f_4}{3}\right] + 2\,O(h)^5$$

which generalises with n intervals to the so-called "extended" Simpson's rule (we assume for convenience of notation that n is even):

$$I = h\left[\frac{f_0}{3} + \frac{2}{3}\sum_{i=1}^{n/2-1} f_{2i} + \frac{4}{3}\sum_{i=1}^{n/2} f_{2i-1} + \frac{f_n}{3}\right] + O\left(\frac{(b-a)^5}{n^4}\right). \tag{3.15}$$

The error term comes from making n errors of $O(h^5)$ (see Eq. (3.13)) with $h = (b-a)/n$

A simple algorithm like Eq. (3.15) does a remarkably good job for many integrals. If you want to be really clever, it is possible to recursively nest extended integration rules inside each other on a set of successively finer subdivisions of the interval to produce a very rapidly converging scheme known as Romberg's method.

3.3.2 Numerical estimation of "improper" integrals

Integration rules like Eq. (3.14) and (3.15) are of no immediately use if the domain of integration is infinite (either $a = \infty$ or $b = \infty$) or if the function$f(x)$ has a singularity on the boundary of the domain of integration. Any such singularity is necessarily integrable. Otherwise the integral would not exist. If a singularity exists at a known point, c, in the interior of the domain then we can split the domain into two ($[a, c]$ and $[c, b]$) so that the singularity is on the boundary. Such integrals are referred to as "improper" integrals.

While one can think of approaches to attempt to estimate such integrals by brute force using the tools above, a much more elegant approach is to use a change of variables which turns the improper integral into a proper one. For example, if the domain of integration is infinite, we can use the change of variables $x = y^{-1}$:

$$\int_a^b f(x)\,dx = \int_{1/b}^{1/a} y^{-2}\,f(y^{-1})\,dy \tag{3.16}$$

which produces a finite domain of integration as either a or b go to infinity.

Now suppose $f(x)$ has a singularity at the lower limit of integration: $f(x) \sim (x-a)^{-\gamma}$ as $x \to a$. We must have $\gamma < 1$ in order for this singularity to be integrable. We can use the change of variables

$$y = (x-a)^{1-\gamma}$$

to obtain the integral

$$I = \int_a^b f(x)\,dx = \int_0^{(b-a)^{1-\gamma}} F(y)\,dy$$

where

$$F(y) = \frac{1}{1-\gamma} y^{\frac{\gamma}{1-\gamma}}\, f\left(y^{\frac{1}{1-\gamma}} + a\right)$$

and $F(y) \to 1$ as $y \to 0$. In both these cases, Eq. (3.14) or (3.15) can be directly applied to the transformed integral. This is an example of a general principle of numerical simulation: if it is possible to use a change of variables to remove explicit infinities from a problem before putting it on a computer then life will likely go much more smoothly.

3.3.3 Equivalence of numerical integration and solution of ordinary differential equations

If there is a singularity at an unknown point in the interior of the domain of integration then we cannot apply any of the above methods. For such problems it is better to go "back to the drawing board" and exploit the fact that computing the integral

$$I = \int_a^b f(x)\,dx$$

is equivalent to solving the differential equation

$$\frac{dy}{dx} = f(x),$$

with initial condition $y(a)$=0. The required integral is $I = y(b)$. Check it yourself. An advantage of this equivalence is that there are powerful general numerical methods available to solve ordinary differential equations which we shall get to later.

Bibliography

[1] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Numerical recipes: The art of scientific computing (3rd ed.). Cambridge University Press, New York, NY, USA, 2007.

[2] Neville's algorithm, 2014. http://en.wikipedia.org/wiki/Neville's_algorithm.

[3] Runge's phenomenon, 2014. http://en.wikipedia.org/wiki/Runge's_phenomenon.

[4] D. Cruz-Uribe and C. J. Neugebauer. An elementary proof of error estimates for the trapezoidal rule. Mathematics Magazine, 76(4):303, 2003. http://www.maa.org/sites/default/files/An_Elementary_Proof30705.pdf.

Notes 4: Root finding algorithms

4.1 Root-finding in one dimension

Given a function, $f(x)$, of a single scalar variable root-finding in one dimension is the task of finding the value or values of x which satisfy the equation

$$f(x) = 0. \tag{4.1}$$

Since most equations cannot be solved analytically, numerical approaches are essential. It is important to remember however that since "most" real numbers do not have exact floating-point representations, we are really interested in finding values of x for which Eq. (4.1) is satisfied approximately in the sense that

$$|f(x)| < \epsilon_{\text{tol}} \tag{4.2}$$

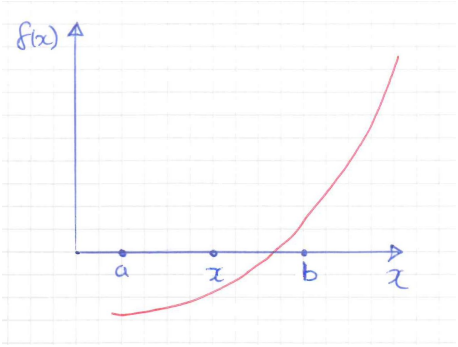
for some pre-specified error tolerance, ϵ_{tol}. All root-finding methods rely on making successive improvements to a reasonable initial guess, x_0, for the position, x_*, of the root. After all, $\mathbb{R}$ is a big place. A root is said to be bracketed by the interval $[a, b]$ if $f(a)f(b) < 0$, that is to say, $f(x)$ has different sign at the two ends of the interval. For continuous functions (and as far as floating-point arithmetic is concerned all functions are continuous), the Intermediate Value Theorem assures the existence of a root somewhere in the interval $[a, b]$. The first step is always to find an interval which brackets the root you are seeking to find. [1] Any reasonable initial guess for the position of the root should be in this interval. Therefore you should always start by plotting your function. This is true for most analytic tasks but especially so for root finding.

There are many algorithms for root finding (and for the related task of finding extremal values fo functions) although we shall only consider a few of them in this module. These algorithms can be usefully divided into two classes: those which use the derivative of the function $f(x)$ and those which do not. Algorithms which use derivatives tend to converge faster but those which do not tend to be more robust. Convergence time is rarely an issue for one-dimensional root finding on modern computers (not necessarily so for higher dimensions) so the increased speed of derivative methods often does not outweigh the sure-footedness of their less sophisticated non-derivative cousins. In any case, it is common to need to analyse functions for which no analytic form for the derivative is known, for example functions which are defined as the outputs of some numerical or combinatorial function.

4.1.1 Derivative-free methods: bracket and bisection method, Brent's method

We start with methods which do not require any knowledge of the derivative. The bracketing-and-bisection algorithm is the most basic root finding method. It involves making successive refinements of the bracketing interval by testing the sign of $f(x)$ at the midpoint of the current interval and then defining a new bracketing interval in the obvious way. At each step, the "best guess" of the position of the root is the midpoint, x:

[1] Note that not every root can be bracketed - a simple counter example is $f(x) = x^2$ which has a root at 0 but no bracketing interval can be chosen. For this reason, the task of finding *all* roots of a nonlinear equation is a-priori a very difficult task.



If we start from a bracketing interval $[a_0, b_0]$, the algorithm consists of the following:

while $b_i - a_i > \epsilon_{\rm tol}$ **do**
  $x = (a_i + b_i)/2.0$;
  if $f(a_i) * f(x) > 0$ **then**
    $a_{i+1} = x$;
    $b_{i+1} = b$;
  else
    $a_{i+1} = a$;
    $b_{i+1} = x$;
  end
end

What is a reasonable value for $\epsilon_{\rm tol}$? A general rule of thumb is to stop when the width of the bracketing interval has decreased to about $(|a_i| + |b_i|)\,\varepsilon_m/2$ where ε_m is the machine precision. You should think about why that is so.

At each step of the algorithm the width, ϵ_i, of the bracketing interval decreases by a factor of two: $\epsilon_{i+1} = \epsilon_i/2$. Hence, $\epsilon_n = \epsilon_0/2^n$. The number of steps needed to reach accuracy $\epsilon_{\rm tol}$ is thus $n = \log_2(\epsilon_0/\epsilon_{\rm tol})$. While the bracketing-and-bisection method is often said to be "slow" it actually converges exponentially fast! Furthermore it cannot fail.

One could try to improve on simple bracketing-and-bisection with a smarter "best guess" of the position of the root at each step. One way to do this is to fit a quadratic, $P(x)$, through the points $(a_i, f(a_i))$, $(x_i, f(x_i))$ and $(b_i, f(b_i))$ where the current interval is $[a_i, b_i]$ and x_i is the current best guess of the position of the root. This quadratic is used to make the next guess, x_{i+1}, of the position of the root by calculating where it crosses zero. That is, we solve $P(x) = 0$. For a quadratic function this can be done analytically. The quadratic formula involves the calculation of (expensive) square roots and generally produces two roots. For these reasons, it is much better in practice to fit x as a function of y and then evaluate the resulting quadratic at $y = 0$ since this results in a unique value of x and does not require the computation of square roots. This is simple but clever work-around goes by the grandiose name of inverse quadratic interpolation.

The Lagrange interpolating polynomial through the points $(f(a_i), a_i)$, $(f(x_i), x_i)$ and $(f(b_i), b_i)$ is obtained from Eq. (3.1):

$$x = \frac{a_i\,[y - f(x_i)][y - f(b_i)]}{[f(a_i) - f(x_i)][f(a_i) - f(b_i)]} + \frac{x_i\,[y - f(a_i)][y - f(b_i)]}{[f(x_i) - f(a_i)][f(x_i) - f(b_i)]} + \frac{b_i\,[y - f(a_i)][y - f(x_i)]}{[f(b_i) - f(a_i)][f(b_i) - f(x_i)]}. \tag{4.3}$$

Evaluating it at $y = 0$ gives the next best guess for the position of the root:

$$x_{i+1} = \frac{a_i\,f(x_i)\,f(b_i)}{[f(a_i) - f(x_i)][f(a_i) - f(b_i)]} + \frac{x_i\,f(a_i)\,f(b_i)}{[f(x_i) - f(a_i)][f(x_i) - f(b_i)]} + \frac{b_i\,f(a_i)\,y - f(x_i)}{[f(b_i) - f(a_i)][f(b_i) - f(x_i)]}. \tag{4.4}$$

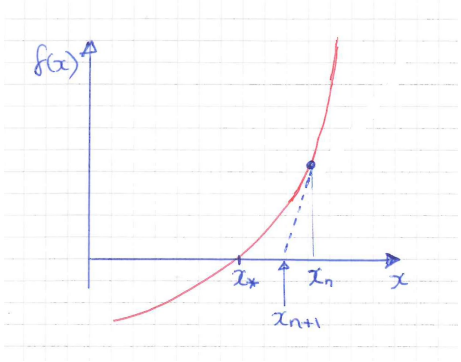
The bracketing interval is then reduced in size as before by testing the sign of $f(x_{i+1})$.

Note that it is possible with this procedure to generate a value for x_{i+1} which lies outside the interval $[a_i, b_i]$. If this happens, one can resort to a simple bisection of the interval as before. Brent's method combines inverse quadratic interpolation with some checks on proposed refinements of the position of the root and some information about previous estimates of the position, using bisection

when necessary, to produce a method which is typically even faster to converge. See [1, chap. 9] for full details. Brent's method is usually the method of choice for generic root-finding problems in one dimension. The idea of inverse quadratic interpolation will come in useful later when we come to the problem of searching for minima.

4.1.2 Derivative methods: Newton-Raphson method

The Newton-Raphson method is the most famous root-finding algorithm. It is a derivative method: it requires the ability to evaluate the derivative $f'(x)$ at any point. The basic idea is the following: at a point, x, which is near to a root, an extrapolation of the tangent line to the curve at x provides an estimate of the position of the root. This is best illustrated geometrically with a picture:



Mathematically, if we are at a point x which is near to a root, x_*, then we wish to find δ such that $f(x+\delta) = 0$. This can be done using Taylor's Theorem, (3.2.1). If x is near to the root then δ is small and we can write

$$0 = f(x+\delta) = f(x) + \delta\, f'(x) + O(\delta^2).$$

Neglecting terms of order δ^2 and above and solving for δ we obtain

$$\delta = -\frac{f(x)}{f'(x)}.$$

Our improved estimate for the position of the root is then $x + \delta$. This process can be iterated. If our current estimate of the position of the root is x_i, then the next estimate is

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}. \tag{4.5}$$

Starting from an initial guess, x_0, the algorithm is the following:

while $|f(x_i)| > \epsilon_{\text{tol}}$ **do**
  $\delta = -\frac{f(x_i)}{f'(x_i)}$;
  $x_{i+1} = x_i + \delta$;
end

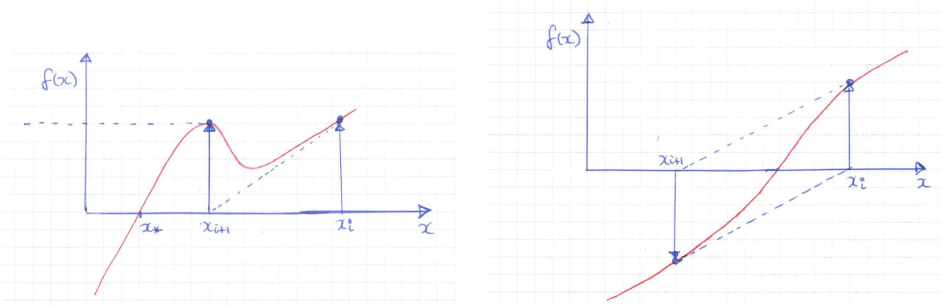
If $f'(x)$ is too difficult (or too expensive) to evaluate analytically for use in Eq. (4.5) then the finite difference formulae developed in Sec. 3.2.1 can be used. This is generally to be discouraged since the additional round-off error (recall Fig. 3.5) can badly degrade the accuracy of the algorithm.

The advantage of Newton-Raphson is that it converges very fast. Let us denote the exact position of the root by x_* and the distance from the root by $\epsilon_i = x_i - x_*$. Using Eq. (4.5, we see that

$$\epsilon_{i+1} = \epsilon_i - \frac{f(x_i)}{f'(x_i)}. \tag{4.6}$$

Since ϵ_i is supposed to be small we can use Taylor's Theorem 3.2.1 :

$$\begin{aligned} f(x_* + \epsilon_i) &= f(x_*) + \epsilon_i f'(x_*) + \frac{1}{2}\epsilon_i^2 f''(x_*) + O(\epsilon_i^3) \\ f'(x_* + \epsilon_i) &= f'(x_*) + \epsilon_i f''(x_*) + O(\epsilon_i^2). \end{aligned}$$



(A) (B)

Figure 4.1: Some "unfortunate" configurations where the Newton-Raphson algorithm runs into trouble.

Using the fact that $x_* + \epsilon_i = x_i$ and $f(x_*) = 0$ we can express the x_i-dependent terms in Eq. (4.6) in terms of ϵ_i and the values of the derivatives of $f(x)$ at the root:

$$\begin{aligned} f(x_i) &= \epsilon_i f'(x_*)\left(1 + \frac{\epsilon_i}{2}\frac{f''(x_*)}{f'(x_*)}\right) + O(\epsilon_i^3) \\ f'(x_i) &= f'(x_*)\left(1 + \epsilon_i \frac{f''(x_*)}{f'(x_*)}\right) + O(\epsilon_i^2) \end{aligned}$$

Substituting these into Eq. (4.6) and keeping only leading order terms in ϵ_i we see that

$$\epsilon_{i+1} = \epsilon_i^2 \frac{f''(x_*)}{2\, f'(x_*)}.$$

Newton-Raphson iteration therefore converges super-exponentially fast.

The disadvantage of the Newton-Raphson method is that, unlike the bracketing-and-bisection method, it is not guaranteed to converge even if you start near to a bracketed root. Fig. 4.1 shows a couple of unfortunate cases where the Newton-Raphson iteration would fail to converge. In Fig. 4.1(A) an iteration of the method happens to hit on an extremum of $f(x)$. Since $f'(x) = 0$ at an extremum, Eq. (4.5) puts the next guess at infinity! In Fig. 4.1(B), a symmetry between successive iterations sends the method into an infinite loop where the iterations alternate between two values either side of the root.

The potential for complicated dynamics should not come as a surprise to those familiar with the basics of dynamical systems. Eq. (4.5) generalises immediately to complex-valued functions. The result is generally a nonlinear iterated map which can lead to very rich dynamics (periodic cycles, chaos, intermittency etc). For example, Fig. 4.2, shows the basins of attraction in the complex plane of the roots of the polynomial

$$f(z) = z^3 - 1 = 0 \tag{4.7}$$

under the dynamics defined by the Newton-Raphson iteration. One of the great pleasures of nonlinear science is the fact that such beauty and complexity can lurk in the seemingly innocuous task of finding the roots of a simple cubic equation!

4.2 Root-finding in higher dimensions

Root finding in higher dimensions means finding solutions of simultaneous nonlinear equations. For example, in two dimensions, we seek values of x and y which satisfy

$$\begin{aligned} f(x,y) &= 0 \\ g(x,y) &= 0. \end{aligned}$$

There is no analogue to the concept of bracketing a root in higher dimensions. This makes the process of root finding in higher dimensions very difficult. In order to make a guess about the location of a root in two dimensions, there is no real substitute for tracing out the zero level curves of each

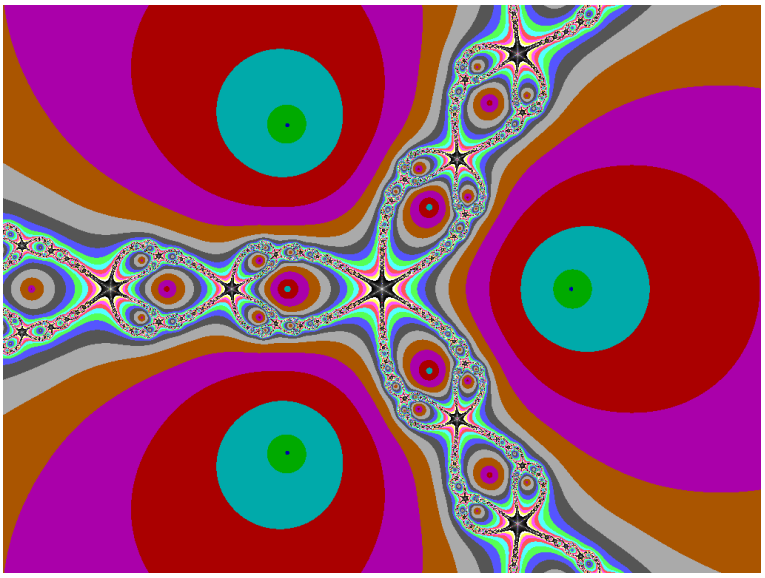


Figure 4.2: (from Wikipedia) Basins of attraction of the roots of Eq. (4.7) in the complex plane. Points are coloured according to the number of iterations of Eq. (4.5) required to reach convergence.

function and seeing if they intersect. In dimensions higher than two, the task rapidly starts to seem like searching for a "needle in a haystack". If, however, one does have some insight as to the approximate location of a root in order to formulate a reasonable initial guess, the Newton-Raphson method does generalise to higher dimensions.

Consider the n-dimensional case. We denote the variables by the vector $\mathbf{x} \in \mathbb{R}^n$. We need n functions of $\mathbf{x}$ to have the possibility of isolated roots. We denote these functions by $F_0(\mathbf{x}) \ldots F_{n-1}(\mathbf{x})$. They can be combined into a single vector valued function, $\mathbf{F}(\mathbf{x})$. We seek solutions of the vector equation

$$\mathbf{F}(\mathbf{x}) = 0. \tag{4.8}$$

If the current estimate of the position of the root is $\mathbf{x}_k$ and we take a step $\boldsymbol{\delta}$ then we can expand the i^{th} component of $F(\mathbf{x}_k + \boldsymbol{\delta})$ using the multivariate generalisation of Taylor's Theorem:

$$F_i(\mathbf{x}_k + \boldsymbol{\delta}) = F_i(\mathbf{x}_k) + \sum_{i=0}^{n-1} J_{ij}(\mathbf{x}_k)\, \delta_j + O(\delta^2) \tag{4.9}$$

where

$$J_{ij}(\mathbf{x}) = \frac{\partial F_i}{\partial x_j}(\mathbf{x})$$

is the (i, j) component of the Jacobian matrix, $\mathbf{J}$, of $\mathbf{F}$ evaluated at $\mathbf{x}$. We wish to choose $\boldsymbol{\delta}$ so that $\mathbf{x}_k + \boldsymbol{\delta}$ is as close as possible to being a root. Setting $F_i(\mathbf{x}_k + \boldsymbol{\delta}) = 0$ in Eq. (4.9) for each i, we conclude that the components of $\boldsymbol{\delta}$ should satisfy the set of linear equations

$$\mathbf{J}(\mathbf{x}_k)\, \boldsymbol{\delta} = \mathbf{F}(\mathbf{x}_k). \tag{4.10}$$

This set of linear equations can be solved using standard numerical linear algebra algorithms:

$$\boldsymbol{\delta} = \text{LinearSolve}(\mathbf{J}(\mathbf{x}_k), \mathbf{F}(\mathbf{x}_k)). \tag{4.11}$$

As for Newton-Raphson iteration in one dimension, the next estimate for the position of the root is $\mathbf{x}_{k+1} = \mathbf{x}_k + \boldsymbol{\delta}$. We have already seen how the two-dimensional case of Newton-Raphson iteration for complex-valued functions can already lead to very nontrivial dynamics. In higher dimensions, the scope for non-convergence becomes even greater. Nevertheless, if one starts close to a root, the above algorithm usually works.

Bibliography

[1] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Numerical recipes: The art of scientific computing (3rd ed.). Cambridge University Press, New York, NY, USA, 2007.

[2] Nelder-mead method, 2014. http://en.wikipedia.org/wiki/Nelder-Mead_method#mediaviewer/File:Nelder_Mead2.gif.

[3] W. Spendley, G. R. Hext, and F. R. Himsworth. Sequential application of simplex designs in optimisation and evolutionary operation. Technometrics, 4(4):441–461, 1962.

[4] J. A. Nelder and R. Mead. A simplex method for function minimization. The Computer Journal, 7(4):308–313, 1965.

[5] S. Singer and J. Nelder. Nelder-mead algorithm. Scholarpedia, 4(7):2928, 2009. http://www.scholarpedia.org/article/Nelder-Mead_algorithm.

[6] R. Fletcher and C. M. Reeves. Function minimization by conjugate gradients. The Computer Journal, 7(2):149–154, 1964.

[7] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain, 1994. http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf.

Notes 5: Numerical linear algebra

5.1 Systems of linear equations

One of the most basic tasks of scientific computing is to find solutions of sets of linear algebraic equations. In general we can have m equations in n unknowns, $x_0, \ldots x_{n-1}$. We write the linear system in matrix-vector form:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b}, \tag{5.1}$$

where

$$\mathbf{A} = \begin{pmatrix} a_{00} & a_{01} & \ldots & a_{0,n-1} \\ a_{10} & a_{11} & \ldots & a_{1,n-1} \\ & \ldots & & \\ a_{m-1,0} & a_{m-1,1} & \ldots & a_{m-1,n-1} \end{pmatrix} \quad \mathbf{x} = \begin{pmatrix} x_0 \\ x_1 \\ \ldots \\ x_{n-1} \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} b_0 \\ b_1 \\ \ldots \\ b_{m-1} \end{pmatrix}$$

There are three cases:

1. $m < n$
   If the number of equations, m, is less than the number of unknowns, n, then the system Eq. (5.1) is said to be under-determined and there is either no solution for $\mathbf{x}$ or multiple solutions spanning a space of dimension less than n known as the nullspace of $\mathbf{A}$.
2. $m > n$
   If the number of equations, m, is greater than the number of unknowns, n, then the system Eq. (5.1) is said to be over-determined and, in general, there is no solution for $\mathbf{x}$. In this case, we are generally interested in solving the linear least squares problem associated with Eq. (5.1) which involves finding the value of $\mathbf{x}$ comes closest to satisfying Eq. (5.1) in the sense that it minimises the sum of the squared errors:

   $$x_* = \arg\min_{\mathbf{x}} |\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|^2 .$$

3. $m = n$
   When the number of equations equals the number of unknowns we generally expect there to exist a single unique solution. This is not guaranteed however unless the rows, or equivalently (since $\mathbf{A}$ is a square matrix) the columns, of $\mathbf{A}$ are linearly independent. If one or more of the m equations is a linear combination of the others, the matrix $\mathbf{A}$ is said to be row-degenerate. If all of the equations contain two or more of the unknowns in exactly the same linear combination, the matrix $\mathbf{A}$ is said to be column-degenerate. Row and column degeneracy are equivalent for square matrices. If $\mathbf{A}$ is degenerate, the linear system Eq. (5.1) is said to be singular. Singular systems effectively have fewer equations (meaning linearly independent equations) than unknowns which puts us back in case 1 above.

From a computational point of view, solving Eq. (5.1) can be a highly non-trivial task, even in the case when $n = m$ and $\mathbf{A}$ is known to be non-degenerate. The reason is again rounding error. It can happen that two equations are sufficiently close to being linearly dependent that round-off errors make them become effectively linearly dependent and a solution algorithm which would find a solution in conventional arithmetic will fail when implemented in floating-point arithmetic. Furthermore, since the solution of Eq. (5.1) involves a large number of additions and subtractions when n is large, round off error can accumulate through the calculation so that the "solution", $\mathbf{x}$, which is found is simply wrong in the sense that when substituted back into Eq. (5.1), $\mathbf{A} \cdot \mathbf{x} - \mathbf{b}$ is significantly different from zero. This is particularly the case for linear systems which are close to being singular in some sense. Such systems are said to be ill-conditioned.

5.2 LU decomposition and applications

5.2.1 Triangular matrices and LU decomposition

An $n \times n$ matrix is lower triangular if all of the entries above the diagonal are zero:

$$\mathbf{L} = \begin{pmatrix} l_{00} & 0 & 0 & \dots & 0 \\ l_{10} & l_{11} & 0 & \dots & 0 \\ l_{20} & l_{21} & l_{22} & \dots & 0 \\ & \dots & & & \\ l_{n-1,0} & l_{n-1,1} & l_{n-1,2} & \dots & l_{n-1,n-1} \end{pmatrix}$$

A matrix is upper triangular if all the entries below the diagonal are zero:

$$\mathbf{U} = \begin{pmatrix} u_{00} & u_{01} & u_{02} & \dots & u_{0,n-1} \\ 0 & u_{11} & u_{12} & \dots & u_{1,n-1} \\ 0 & 0 & u_{22} & \dots & u_{2,n-1} \\ & \dots & & & \\ 0 & 0 & 0 & \dots & u_{n-1,n-1} \end{pmatrix}$$

Triangular matrices are important because linear systems involving such matrices are particularly easy to solve. If $\mathbf{U}$ is upper triangular, then the linear system

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{b} \tag{5.2}$$

can be solved by a simple iterative algorithm known as back substitution:

$x_{n-1} = \frac{b_{n-1}}{u_{n-1,n-1}}$;
for $i = n-2$ **to** 0 **do**

$$x_i = \frac{1}{u_{i,i}} \left[b_i - \sum_{j=i+1}^{n-1} u_{i,j}\, x_j \right] ;$$

end

Similarly, if $\mathbf{L}$ is lower triangular, the linear system

$$\mathbf{L} \cdot \mathbf{x} = \mathbf{b} \tag{5.3}$$

can be solved by a simple iterative algorithm known as forward substitution:

$x_0 = \frac{b_0}{l_{0,0}}$;
for $i = 1$ **to** $n-1$ **do**

$$x_i = \frac{1}{l_{i,i}} \left[b_i - \sum_{j=0}^{i-1} l_{i,j}\, x_j \right] ;$$

end

Suppose we can find a way to write a non-triangular matrix, $\mathbf{A}$ in the form

$$\mathbf{A} = \mathbf{L} \cdot \mathbf{U} \tag{5.4}$$

where $\mathbf{L}$ is lower triangular and $\mathbf{U}$ is upper triangular. This is referred to as an LU decomposition of $\mathbf{A}$. We can then solve the linear system Eq. (5.1) as follows:

- Define $\mathbf{y} = \mathbf{U} \cdot \mathbf{x}$
- Solve the linear system $\mathbf{L} \cdot \mathbf{y} = \mathbf{b}$ by forward substitution to obtain $\mathbf{y}$.
- Solve the linear system $\mathbf{U} \cdot \mathbf{x} = \mathbf{y}$ by back substitution to obtain $\mathbf{x}$.

Note that $\mathbf{L}$ and $\mathbf{U}$ as written above each have $\frac{1}{2}n(n+1)$ non-zero entries. The total number of unknowns in the product $\mathbf{L} \cdot \mathbf{U}$ is therefore $n^2 + n$ whereas there are only n^2 equations implied by Eq. (5.4). There is therefore some freedom to introduce additional constraints on the values of the entries of $\mathbf{L}$ and $\mathbf{U}$ to reduce the number of unknowns. It is common to impose that either $\mathbf{L}$ or $\mathbf{U}$

should be unit triangular, meaning that the n diagonal elements are all equal to 1. We shall adopt the convention that $\mathbf{L}$ is unit triangular, i.e. $l_{ii} = 1$ for $i = 0, \ldots, n-1$.

It turns out, however, that many non-singular square matrices do not have an LU decomposition. The reason is already evident in a simple example 3×3 example. Let us try to find an LU decomposition of a general 3×3 matrix (assumed to be non-singular) by writing

$$\begin{pmatrix} l_{00} & 0 & 0 \\ l_{10} & l_{11} & 0 \\ l_{20} & l_{21} & l_{22} \end{pmatrix} \begin{pmatrix} u_{00} & u_{01} & u_{02} \\ 0 & u_{11} & u_{12} \\ 0 & 0 & u_{22} \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}.$$

Upon multiplication we see that the first entry gives $a_{00} = l_{00}\, u_{00}$. If it so happens that $a_{00} = 0$ then we would have to choose either $l_{00} = 0$ or $u_{00} = 0$. The first choice would result in $\mathbf{L}$ being singular and the second choice would result in $\mathbf{U}$ being singular. The product $\mathbf{L} \cdot \mathbf{U}$ is then necessarily singular but this is impossible since we have assumed that $\mathbf{A}$ is non-singular. Hence the matrix $\mathbf{A}$ does not have an LU decomposition. We could get around this problem by swapping the first row of $\mathbf{A}$ with either the second or the third row so that the upper left entry of the resulting matrix is nonzero. This must be possible because $\mathbf{A}$ is nonsingular so all three of a_{00}, a_{10} and a_{20} cannot simulataneously be zero. If the same problem is encountered in subsequent steps, it can be removed in the same way. We conclude that if the matrix $\mathbf{A}$ does not have an LU decomposition, there is a permutation of the rows of $\mathbf{A}$ which does.

There is a general principle at work here. It turns out that for every nondegenerate square matrix, there exists a permutation of the rows which does have an LU decomposition. Such a permutation of rows corresponds to left multiplication by a permutation matrix, $\mathbf{P}$, which is a matrix obtained by permuting the columns of an $n \times n$ identity matrix. From the point of view of solving linear systems, multiplication by $\mathbf{P}$ makes no difference provided that we also multiply the righthand side, $\mathbf{b}$, by $\mathbf{P}$. We are simply writing the equations in a different order: $\mathbf{P} \cdot \mathbf{A} \cdot \mathbf{x} = \mathbf{P} \cdot \mathbf{b}$. The decomposition

$$\mathbf{P} \cdot \mathbf{A} = \mathbf{L} \cdot \mathbf{U}$$

is known as LU decomposition with partial pivoting and can be used to solve general nonsingular systems of linear equations.

5.2.2 Crout's algorithm for LU decomposition with partial pivoting

Having established that the LU decomposition of a matrix is useful, let us now consider how to actually compute it. The basic procedure is known as Crout's algorithm. It works by writing out all n^2 terms in the product in Eq. (5.4) in a special order so that the nonzero values of u_{ij} and l_{ij} can be explicitly solved for using a procedure similar to the back and forward substitution algorithms described above.

Let us first look at the (i, j) entry of the product in Eq. (5.4) which gives a_{ij}. It is obtained by taking the dot product of row i of $\mathbf{L}$ with column j of $\mathbf{U}$. Row i of $\mathbf{L}$ has i nonzero elements:

$$(l_{i0}, l_{i1} \ldots l_{i,i-1}, l_{i,i}, 0, \ldots 0).$$

Column j of $\mathbf{U}$ has j nonzero elements:

$$(u_{0j}, u_{1j} \ldots u_{j-1,j}, u_{j,j}, 0, \ldots 0).$$

When we take the dot product there are three cases:

1. $\mathbf{i < j}$: In this case we have

$$a_{ij} = l_{i0}u_{0j} + l_{i1}u_{1j} + \ldots + l_{i,i-1}u_{i-1,j} + l_{ii}u_{ij}$$

Using the fact that we have adopted the convention that $l_{ii} = 1$ we can write

$$u_{ij} = a_{ij} - \sum_{k=0}^{i-1} l_{ik}u_{kj}. \tag{5.5}$$

2. $\mathbf{i = j}$: Actually this is the same as case 1 with $j = i$. We have

$$a_{ii} = l_{i0}u_{0i} + l_{i1}u_{1i} + \ldots + l_{i,i-1}u_{i-1,i} + l_{ii}u_{ii},$$

which gives

$$u_{ii} = a_{ii} - \sum_{k=0}^{i-1} l_{ik}u_{ki}. \tag{5.6}$$

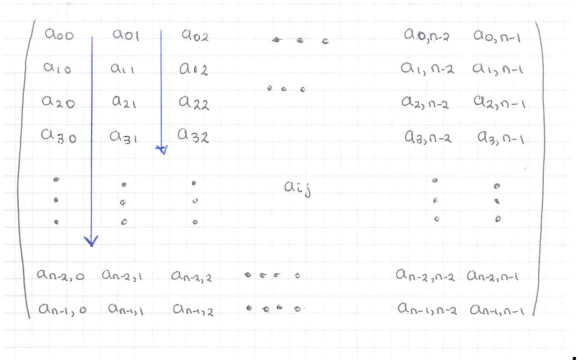
3. $\mathbf{i > j}$: In this case we have

$$a_{ij} = l_{i0}u_{0j} + l_{i1}u_{1j} + \ldots + l_{i,j-1}u_{j-1,j} + l_{ij}u_{jj}.$$

This gives an equation for l_{ij}:

$$l_{ij} = \frac{1}{u_{jj}}\left(a_{ij} - \sum_{k=0}^{j-1} l_{ik}u_{kj}\right). \tag{5.7}$$

At first sight, Eqs. (5.5), (5.6) and (5.7) do not appear to get us any further towards our goal since the unknown l_{ij} and u_{ij} appear on both sides of each. This is where the bit about writing the equations in the correct order comes in. To begin with, let us assume that no pivoting is required. Crout's algorithm works its way through the matrix $\mathbf{A}$ starting at the top left element a_{00}, traversing the row index first and then incrementing the column index:

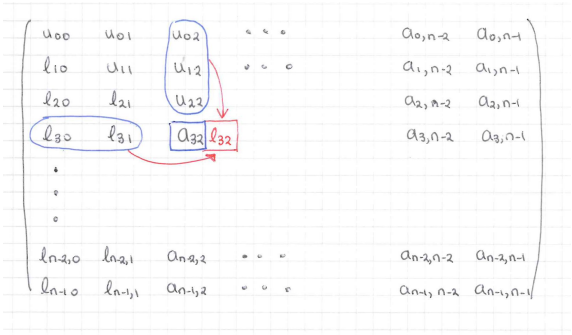


.

As it progresses it computes as follows:

Begin by setting the l_{ii} to 1
for $i = 0$ **to** $n-1$ **do**
  $l_{ii} = 0$;
end
First loop over columns
for $j = 0$ **to** $n-1$ **do**
  Loop over rows in three stages
  Stage 1
  for $i = 0$ **to** $j-1$ **do**
    Since $j < i$ we use Eq. (5.5) to compute u_{ij}
    $$u_{ij} = a_{ij} - \sum_{k=0}^{i-1} l_{ik}u_{kj};$$
    $a_{ij} = u_{ij}$;
  end
  Stage 2: i=j;
  Since $i = j$ we use Eq. (5.6) to compute u_{ii}
  $$u_{ii} = a_{ii} - \sum_{k=0}^{i-1} l_{ik}u_{ki};$$
  $a_{ii} = u_{ii}$;
  Stage 3
  for $i = j+1$ **to** $n-1$ **do**
    Since $j > i$ we use Eq. (5.7) to compute l_{ij}
    $$l_{ij} = \frac{1}{u_{jj}}\left(a_{ij} - \sum_{k=0}^{j-1} l_{ik}u_{kj}\right);$$
    $a_{ij} = l_{ij}$;
  end
end

Here is the state of the array $\mathbf{A}$ after this procedure has reached the element a_{32}:



.

Notice how the entries of $\mathbf{L}$ and $\mathbf{U}$ required to compute a_{32} have already been computed by the time the algorithm reaches a_{32}. This is true for all entries $a_{i,j}$. Notice also that the value of a_{32} is never needed again since it is only required in the computation of l_{32}. This is also true for all entries $a_{i,j}$. This is why as the algorithm works its way through the array, it can over-write the successive values of a_{ij} with the computed value of l_{ij} or u_{ij} as shown. Crout's algorithm is an example of an in-place algorithm. It does not need additional storage to calculate its output and returns the output in the same physical array as its input. After completion, the array $\mathbf{A}$ looks as follows:

$$\mathbf{A} = \begin{pmatrix} u_{00} & u_{01} & u_{02} & \dots & u_{0,n-1} \\ l_{10} & u_{11} & u_{12} & \dots & u_{1,n-1} \\ l_{20} & l_{21} & u_{22} & \dots & u_{2,n-1} \\ & \dots & & & \\ l_{n-1,0} & l_{n-1,1} & l_{n-1,2} & \dots & l_{n-1,n-1} \end{pmatrix}.$$

Remember that the diagonal entries of $\mathbf{L}$ are not stored since we have adopted the convention that $\mathbf{L}$ is unit triangular. The only question which remains is how to incorporate pivoting. The need for pivoting is evident from the equation for l_{ij} in the algorithm. Here there is a trick which is explained in [1, chap. 2] but looks suspiciously like magic to me.

Note that the LU decomposition contains 3 nested loops. It is therefore an $O(n^3)$ algorithm.

5.2.3 Other applications of LU decomposition: matrix inverses and determinants

1. **Computing the inverse of a matrix**
   In the vast majority of cases when a linear system like Eq. (5.1) appears, we are interested in finding $\mathbf{x}$. This is much more efficiently done using the algorithm described above rather than by explicitly computing $\mathbf{A}^{-1}$ and then computing the product $\mathbf{A}^{-1} \cdot \mathbf{b}$. Nevertheless on the rare few occasions when you do really need $\mathbf{A}^{-1}$, the LU decomposition can be used to compute it. To see how to do it note that the LU decomposition can also be used to solve matrix equations of the form
   $$\mathbf{A} \cdot \mathbf{X} = \mathbf{B} \tag{5.8}$$
   where $\mathbf{X}$ and $\mathbf{B}$ are $n \times m$ matrices. The case $m = 1$ corresponds to Eq. (5.1). The trick is to realise that once the LU decomposition has been computed, we can use the forward and back substitution algorithm to solve a sequence of m linear systems containing the successive columns of $\mathbf{B}$ as their right hand sides. The results will be the columns of $\mathbf{X}$. To obtain $\mathbf{A}^{-1}$ we take $\mathbf{B}$ to be an $n \times n$ identity matrix. The matrix $\mathbf{X}$ obtained in this case is then the inverse of $\mathbf{A}$.
2. **Computing the determinant of a matrix**
   Determinants of matrices are easy to compute from the LU decomposition by observing that the determinant of a triangular matrix is just the product of the diagonal elements. If we have an LU decomposition $\mathbf{P} \cdot \mathbf{A} = \mathbf{L} \cdot \mathbf{U}$ in which we have arranged for $\mathbf{L}$ to be unit triangular, then
   $$\det(\mathbf{A}) = \det(\mathbf{P}^{-1}) \det(\mathbf{L}) \det(\mathbf{U}) = (-1)^s \left(\prod_{i=0}^{n-1} u_{ii} \right) \tag{5.9}$$
   where s is the number of row exchanges in the permutation matrix $\mathbf{P}$.

5.2.4 Sparse linear systems

A linear system is called sparse if only a small number of is matrix elements, a_{ij}, are nonzero. Here "small" usually means $O(n)$. Sparse matrices occur very frequently: some of the most common occurences are in finite difference approximations to differential equations and adjacency matrices for networks with low average degree. An archtypal example of a sparse matrix is a a tri-diagonal matrix in which the only non-zero elements are on the diagonal plus or minus one column:

$$\mathbf{A} = \begin{pmatrix} d_0 & a_0 & 0 & 0 & \dots & & 0 \\ b_1 & d_1 & a_1 & 0 & \dots & & 0 \\ 0 & b_2 & d_2 & a_2 & \dots & & 0 \\ 0 & 0 & b_3 & d_3 & \dots & & 0 \\ & \dots & & & & & \\ 0 & 0 & 0 & \dots & b_{n-2} & d_{n-2} & a_{n-2} \\ 0 & 0 & 0 & \dots & 0 & b_{n-1} & d_{n-1} \end{pmatrix}. \tag{5.10}$$

It is very inefficient to use methods designed for general matrices to solve sparse problems since most of your computer's memory will be taken up with useless zeros and most of the $O(n^3)$ operations required to solve a linear system for example would be trivial additions and multiplications of zeroes. Using the LU decomposition algorithm of Sec. 5.2, dense matrices with n of the order hundreds can be routinely solved (assuming that they are not too close to being singular). As n starts to get of order 1000, computational time starts to become the limiting factor. For sparse matrices, however values of n in the tens of thousands can be done routinely and calculations with n in the millions can be tackled without difficulty on parallel machines. Such matrices could not even be stored in dense formats. It is therefore very important to exploit sparsity when it is present in a problem

As an example, calculation of the matrix-vector product $\mathbf{A} \cdot \mathbf{x}$ requires $O(2n^2)$ operations. For the tridiagonal matrix in Eq. (5.10), it is clear that $\mathbf{A} \cdot \mathbf{x}$ can be computed in about $O(3n)$ operations. Storing the matrix also requires only $O(3n)$ doubles as opposed to n^2 for an equivalently sized dense matrix.

Bibliography

[1] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Numerical recipes: The art of scientific computing (3rd ed.). Cambridge University Press, New York, NY, USA, 2007.

Notes 6: Optimisation

6.1 Local optimisation in one dimension

Given a real-valued function, $f(x)$, of a single scalar variable, optimization is the task of finding values of x for which $f(x)$ is extremal. That is to say, we seek values of x at which $f'(x) = 0$. These correspond to maxima and minima. It is sufficient to consider only methods for finding minima since maxima can be found be finding minima of $-f(x)$. General optimisation problems can be divided into two classes: local and global. The distincion rests with whether we are required to find local or global extrema of the function to be optimised. As we shall see, local optimisation is relatively easy. Global optimisation on the other hand is usually very hard. In one-dimension there is a simple procedure for finding local minima (which actually generalises quite well to higher dimensions): evaluate the local gradient and "go downhill" until you can't go downhill any more (to within a pre-specified error tolerance of course). If a local minimum exists such a procedure cannot fail to find it.

6.1.1 Golden section search - a bracketing-and-bisection method

Let us begin by adapting the bracketing-and-bisection method of finding roots of $f(x)$ to the problem of finding local minima. In one dimension, a root is said to be bracketed by the ordered triple (a, b, c) if $f(b) < f(a)$ and $f(b) < f(c)$. "Ordered" means that $a < b < c$. Starting from a bracketing triple (a_0, b_0, c_0), the idea is to generate successive refinements of the bracketing triple (a, b, c) until $c - b < \epsilon_{\rm tol}$ where $\epsilon_{\rm tol}$ is the pre-specified error tolerance. One refinement strategy would be to evaluate $f(x)$ at the midpoints, x_1 and x_2 of the two respective subintervals, (a, b) and (b, c), test which of the sub-intervals (a, x_1, b), (x_1, b, x_2) and (b, x_2, c) is itself a bracketing subinterval and then set the refinied interval equal to the one which passes this test. This would reduce the bracketing interval by a (fixed) factor of $1/2$ at each step but requires two evaluations of the function $f(x)$ per step. It turns out that it is possible to devise a refinement algorithm which requires only a single evaluation of $f(x)$ per step which retains the attractive property of reducing the bracketing interval by a fixed amount per step. This algorithm is known as the golden section search. The price to be paid for fewer function evaluations is that we can no longer reduce the size of the bracketing interval by a factor of $1/2$ at each step but only by a factor of $(1 + \sqrt{5})/2 \approx 0.618$. The efficiency savings in the reduced number of function evaluations usually more than compensate for the slightly slower rate of interval shrinkage (the convergence is still exponentially fast).

The idea of the golden section search is as follows. At each successive refinement, we pick a new point, x, in the larger of the two intervals (a, b) and (b, c), evaluate $f(x)$ and use it to decide how to shrink the interval. The possible options are sketched in Fig. 6.1. Let us suppose that we choose x to be in the interval (b, c) (Case 1 in Fig. 6.1). After evaluation of $f(x)$ the new bracketing triple is either (a, b, x) or (b, x, c). The size of the new bracketing triple will be either $x - a$ or $c - b$. We require these two lengths to be equal. The reasoning behing this is the following: if they were not not equal, the rate of convergence could be slowed down in the case of a run of bad luck which causes the longer of the two to be selected consecutively. Therefore we chose

$$x - a = c - b \Rightarrow x = a - b + c. \tag{6.1}$$

Note that

$$\begin{aligned} b < x \quad &\Rightarrow \quad b < a - b + c \\ &\Rightarrow \quad b - a < c - b \\ &\Rightarrow \quad x \text{ is in the larger of the two sub-intervals.} \end{aligned}$$

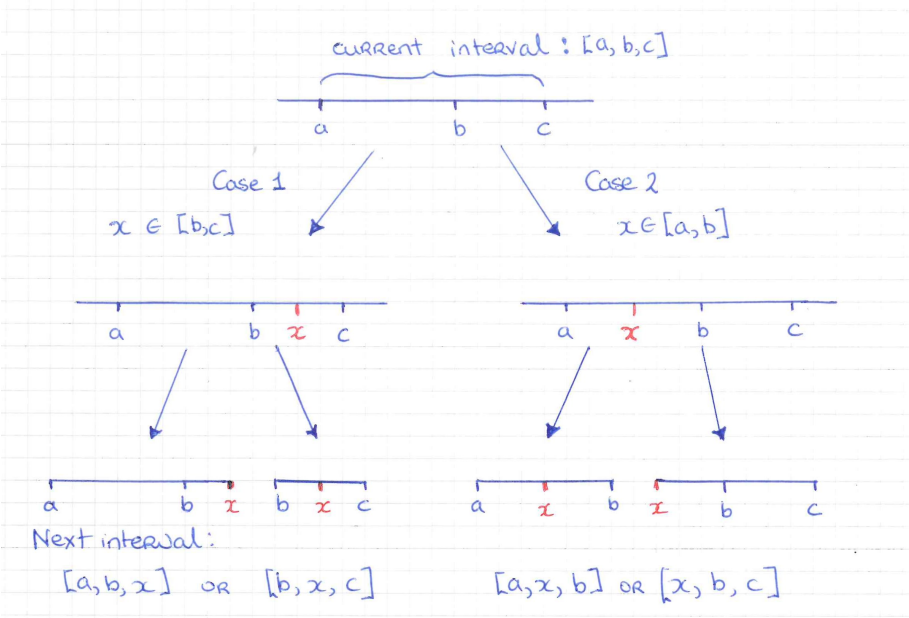


Figure 6.1: Map of possible refinements of a bracketing triple (a, b, c) in the context of the golden section search algorithm for finding a minimum of a function of a single variable.

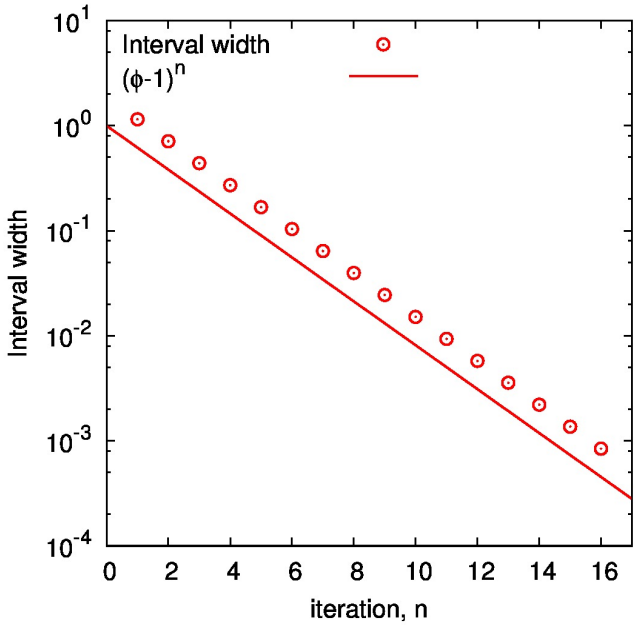


Figure 6.2: Interval width as a function of number of iterations, n, for Golden Section search applied to the function $f(x) = x^{-1}\, e^x$ which has a local minimum at $x = 1$. The theoretical convergence rate, $(1 - \phi)^n$, (where $\phi = (1 + \sqrt{5})/2$ is the Golden Mean) is shown by the solid line.

It remains to decide where to choose x within the interval (b, c). A naive suggestion would be to choose x to be the midpoint of (b, c). This, it turns out, would destroy the property that the interval shrinks by a fixed factor at each iteration. We characterise the "shape" of a bracketing triple by the ratio

$$w = \frac{c-b}{c-a} \tag{6.2}$$

which measures the proportion of the bracketing interval filled up by the second sub-interval. Obviously the proportion filled by the first sub-interval is

$$1 - w = \frac{b-a}{c-a}. \tag{6.3}$$

In order to ensure that the interval shrinks by a fixed factor at each iteration, the geometry of the refined triple should be the same as that of the current triple. Therefore if the new interval is (b, x, c), we require that

$$\frac{c-x}{c-b} = w.$$

Using Eqs. (6.1), (6.2) and (6.3) we get

$$w = \frac{c-(a-b+c)}{c-b} = \frac{b-a}{c-b} = \frac{\frac{b-a}{c-a}}{\frac{c-b}{c-a}} = \frac{1-w}{w}. \tag{6.4}$$

Rearranging this gives a quadratic equation or the shape ratio, w:

$$w^2 + w - 1 = 0 \quad \Rightarrow w = \frac{\sqrt{5}-1}{2}. \tag{6.5}$$

The appearance of the so-called "Golden mean", $\phi = (\sqrt{5}+1)/2$, in this equation gives the method its name. You might ask what happens if the new interval had been (a, b, x)? It is almost the same. If we define the shape ratio to be

$$w = \frac{x-b}{x-a},$$

we get the quadratic $w^2 - 2w + 1 = 0$ which has only the single root $w = 1$. This is not a valid outcome. However, we can simply swap the order of the large and small subinterval by requiring that

$$w = \frac{b-a}{x-a},$$

and this again returns the Golden mean, Eq. (6.5).

We must not forget that we could have chosen x to be in the interval (a, b) (Case 2 in Fig. 6.1). In this case, a similar argument (left as an exercise) shows that we need $b - a > c - b$ (i.e. x is again in the larger of the two subintervals) and the new interval is either (a, b, x) or (b, x, c) and the shape ratio w which preserves the geometry is again the Golden mean, Eq. (6.5). Gathering together these findings, the algorithm for finding the minimum is the following:

$w = (\sqrt{5}-1)/2$;
while $c - a > \epsilon_{\rm tol}$ **do**
  if $|c-b| > |b-a|$ **then**
    $x = b + (1-w)(c-b)$;
    if $f(b) < f(x)$ **then**
      $(a,b,c) = (a,b,x)$;
    else
      $(a,b,c) = (b,x,c)$;
    end
  else
    $x = b - (1-w)(b-a)$;
    if $f(b_i) < f(x)$ **then**
      $(a,b,c) = (x,b,c)$;
    else
      $(a,b,c) = (a,x,b)$;
    end
  end
end

6.1.2 Optimisation by parabolic interpolation

The golden section search is in a sense a worst case algorithm which assumes nothing about the function being minimised except that it has a minumum. In many situations, the function $f(x)$ is continuous and differentiable. In this case, the function can be well approximated near its minimum by a second order Taylor expansion which is parabolic in nature. Given a bracketing triple, (a,b,c), one can fit a quadratic through the three points $(a, f(a))$, $(b, f(b))$ and $(c, f(c))$. By analytically calculating the point at which this parabola reaches its minimum one can step to the minimum of the function (or very close to it) in a single step. Using Eq. (3.1) one can show that the value of x for which this parabola is a minimum is

$$x = b - \frac{1}{2}\frac{(b-a)^2\left[f(b)-f(a)\right] - (b-c)^2\left[f(b)-f(a)\right]}{(b-a)\left[f(b)-f(c)\right] - (b-c)\left[f(b)-f(a)\right]}. \tag{6.6}$$

The value of $f(x_*)$ can then be used, as in the Golden section search to define a refined bracketing triple. This is referred to as parabolic minimisation and can yield very fast convergence for smooth functions. In practice, Eq. (6.6) can be foiled if the points happen to become collinear or happen to hit upon a parabolic maximum. Therefore parabolic minimisation algorithms usually need to perform additional checking and may resort to bisection if circumstances require.

6.2 Local optimisation in higher dimensions

Searching for local extrema in higher dimensional spaces is obviously more difficult than in one dimension but it does not suffer from the general intractability of higher dimensional root-finding which we mentioned in Sec. 4.2. The reason is because the concept of "going downhill" generalises to higher dimensions via the local gradient operator. Therefore, one always has at least the possibility to take steps in the right direction. Higher dimension optimisation algorithms can be grouped according to whether or not they require explicit evaluation of the local gradient of the function to be minimised.

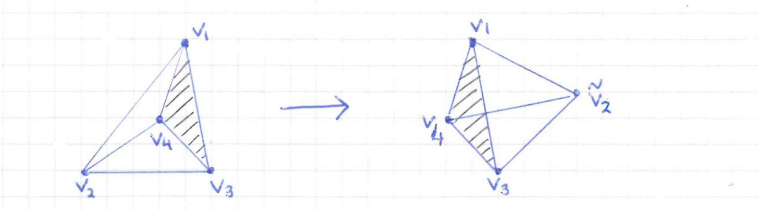
6.2.1 A derivative-free method: Nelder-Mead downhill simplex algorithm

Computing derivatives of multivariate functions can be complicated and expensive. The Nelder-Mead algorithm is a derivative-free method for nonlinear multivariate optimisation which works remarkably well on a large variety of problems, including non-smooth functions (since it doesn't require derivatives). It is conceptually and geometrically very simple to understand and to use even if it can be a bit slow. Everyone should know about this algorithm!

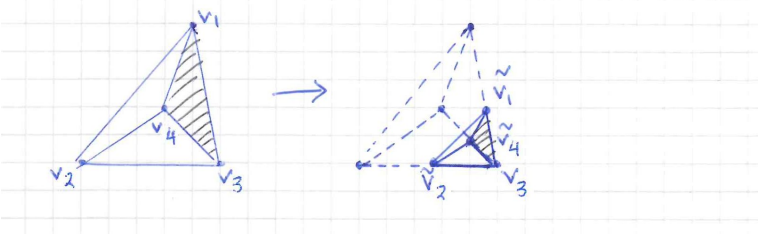
A simplex is a special polytope of N + 1 vertices in N dimensions. Examples of simplices include a line segment on a line, a triangle on a plane, a tetrahedron in three-dimensional space and so forth. The idea of the Nelder-Mead algorithm is to explore the search space using a simplex. The

function to be minimised is evaluated on each of the vertices of the simplex. By comparing the values of the function on the vertices, the simplex can get a sense of which direction is downhill even in high dimensional spaces. At each step in the algorithm is uses some geometrical rules to generate test points, evaluates the objective function at these test points and uses the results to define a new simplex which has moved downhill with respect to the original. For this reason the algorithm is often called the downhill simplex algorithm or the amoeba method. The idea is best conveyed from a movie: see [2]

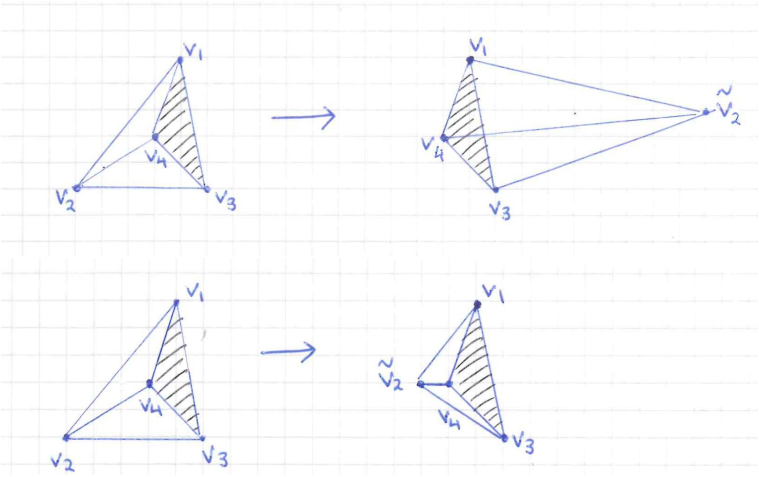
The simplest way to generate test points is to replace the worst point with a point reflected through the centroid of the remaining N points. If this point is better than the best current point then we throw away the previous worst point and update with the new one:



This reflects the simplex through the centroid in a direction which goes downhill. In this picture, and those below, we assume that the highest value of f is attained on the vertex v_2 and the lowest value of f is attained on the vertex v_3. The vertices of the new simplex which have changed during a move are denoted with tildes. If the point obtained by reflection of the worst vertex through the centroid isn't much better than the previous value then we shrink the simplex towards the best point:



These two moves, which preserve the shape of the simplex, constituted the earliest variant of the downhill simplex method designed by Spendley, Hext and Himsworth in [3]. Nelder and Mead added two additional moves [4], expansion and contraction:



These additional moves allow the simplex to change shape and to "squeeze through" narrow holes. The algorithm is stopped when the volume of the simplex gets below a predefined threshold, ϵ_{tol}. For a full discussion of the method see [1, chap. 10] or [5].

6.2.2 A derivative method: the conjugate gradient algorithm

In this section we consider methods for minimising $f(\mathbf{x})$ with $\mathbf{x} = (x_1, x_2 \dots x_n) \in \mathbb{R}^n$ which make use of the gradient of $f(\mathbf{x})$. That is, we can evaluate $\nabla f(\mathbf{x})$ at any point $\mathbf{x}$. Recall that the gradient

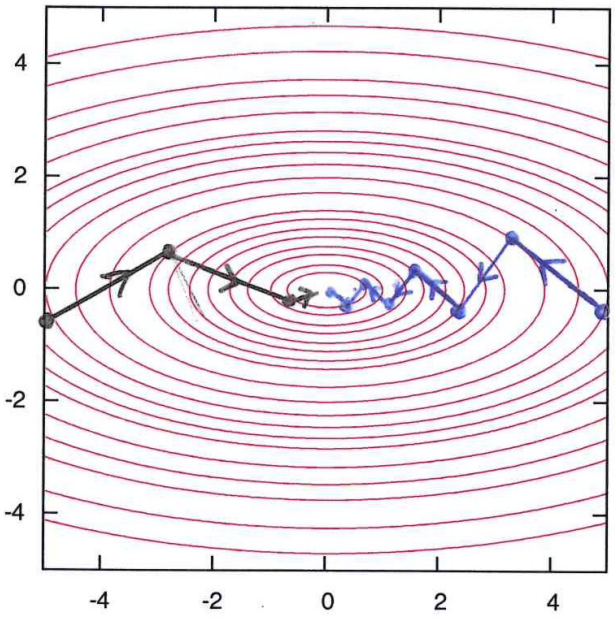


Figure 6.3: Path followed by the Method of Steepest Descent (right path) and Conjugate Gradient Method (left path) during the minimisation of a quadratic function in two dimensions containing a long narrow(ish) valley.

operator is simply the vector of partial derivatives:

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}(\mathbf{x}), \frac{\partial f}{\partial x_2}(\mathbf{x}), \dots \frac{\partial f}{\partial x_n}(\mathbf{x}) \right).$$

The gradient of f at $\mathbf{x}$ points in the direction in which f increases most quickly. Therefore $-\nabla f(\mathbf{x})$ points in the direction in which f decreases most quickly. The ability to calculate the gradient therefore means that we always know which direction is "downhill".

The concept of line minimisation allows us to think of minimisation problems in multiple dimensions in terms of repeated one-dimensional minimisations. Given a point $\mathbf{x}$ and a vector $\mathbf{u}$, the line minimiser of f from $\mathbf{x}$ in the direction $\mathbf{u}$ is the point $x_* = \mathbf{x} + \lambda_{\min} \mathbf{u}$ where

$$\lambda_{\min} = \arg\min_{\lambda} f(\mathbf{x} + \lambda \mathbf{u}).$$

The important point is that this is a one-dimensional minimisation over λ which can be done, for example, using the golden section search described in Sec. 6.1. Evaluating $f(\mathbf{x} + \lambda_{\min} \mathbf{u})$ gives the minimum value of f along the line in $\mathbb{R}^n$ parameterised by $\mathbf{x} + \lambda \mathbf{u}$. A potential strategy for minimising a function in $\mathbb{R}^n$ is to perform sequential line minimisations in different directions until the value of $f(x)$ stops decreasing. This strategy will work but for it to work well we need to find a "good" set of directions in which to perform the line minimisations.

How should we choose the next direction after each line minimisation? This is where being able to compute the gradient is very useful. Firstly we remark that if $\mathbf{x}_*$ is a line minimiser of f in the direction $\mathbf{u}$, then the gradient of f at $\mathbf{x}_*$, $\nabla f(\mathbf{x}_*)$, must be perpendicular to $\mathbf{u}$. If this were not the case then the directional derivative of f in the direction of $\mathbf{u}$ at xv_* would not be zero and $\mathbf{x}_*$ would not be a line minimum. Therefore, after we arrive at a line minimum, $\mathbf{x}_*$, $-\nabla f(\mathbf{x}_*)$, points directly downhill from $\mathbf{x}_*$ and provides a natural choice of direction for the next line minimisation. Iterating this process gives a method known as the Method of Steepest Descent.

While the Method of Steepest Descent works well for many functions it can be slow to converge. The problem with this method is visualised in Fig. 6.3. Since the method is forced to make right-angled turns, it can be very inefficient at making its way to the bottom of narrow valleys. This is illustrated by the (blue) path on the right side of Fig. 6.3. It would be better to allow the algorithm to make steps in directions which are not parallel to the local gradient as in the left path in Fig. 6.3. We are then back to our original question of how to choose the direction for the next step. It turns out that a very good choice is to select the new direction in such a way that the infinitessimal change in the gradient is parallel to the local gradient (or perpendicular to the direction along which we have just minimised). This leads to a method known as the Conjugate Gradient Algorithm.

Underlying this algorithm is the assumption that the function to be approximated can be approximated by a quadratic form centred on some (fixed) point $\mathbf{p}$. The point $\mathbf{p}$ should be thought of as the origin of the coordinate system. Before discussing the method in detail, let us explore some consequences of this assumption. The approximating quadratic form is determined by the multivariate Taylor's Theorem:

$$\begin{aligned} f(\mathbf{p}+\mathbf{x}) &\approx f(\mathbf{p}) + \sum_{i=0}^{n-1} x_i \frac{\partial f}{\partial x_i}(\mathbf{p}) + \frac{1}{2}\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} x_i x_j \frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{p}) \\ &= c - \mathbf{b}\cdot\mathbf{x} + \frac{1}{2}\mathbf{x}\cdot\mathbf{A}\cdot\mathbf{x}, \end{aligned} \tag{6.7}$$

where

$$\mathbf{b} = -\nabla f(\mathbf{p})$$

and the matrix $\mathbf{A}$ has components

$$A_{ij} = \frac{\partial f}{\partial x_i \partial x_j}(\mathbf{p}).$$

$\mathbf{A}$ is known as the Hessian matrix of f at $\mathbf{p}$. To the extent that the approximation in Eq. (6.7) is valid, the gradient of f at $\mathbf{x}$ is

$$\nabla f(\mathbf{p}+\mathbf{x}) = \mathbf{A}\cdot\mathbf{x} - \mathbf{b}. \tag{6.8}$$

The infinitessimal variation in the gradient as we move in some direction is therefore

$$\delta(\nabla f) = \mathbf{A}\cdot(\delta\mathbf{x}). \tag{6.9}$$

Let us now return to the discussion of how to choose directions for line minimisation. Suppose that we have just done a minimisation in the direction $\mathbf{u}$. We wish to choose a new direction $\mathbf{v}$ such that the change in gradient upon going in the direction $\mathbf{v}$ is perpendicular to $\mathbf{u}$. From Eq. (6.9), we see that

$$\begin{aligned} \mathbf{u}\cdot\delta(\nabla f) &= 0 \\ \Rightarrow \mathbf{u}\cdot\mathbf{A}\cdot(\delta\mathbf{x}) &= 0. \end{aligned}$$

Hence $\mathbf{v}$ should point in the direction of this variation $\delta(\mathbf{x})$ such that

$$\mathbf{u}\cdot\mathbf{A}\cdot\mathbf{v} = 0. \tag{6.10}$$

Vectors $\mathbf{u}$ and $\mathbf{v}$ satisfying this condition are said to be conjugate with respect to the Hessian matrix, $\mathbf{A}$. Note the difference with the Method of Steepest Descent which selected $\mathbf{v}$ such that $\mathbf{u}\cdot\mathbf{v} = 0$. The conjugate gradient method constructs a sequence of directions, each of which is conjugate to all previous directions and solves the line minimisations analytically along these direction (this is possible since we are minimising along a section of a quadratic form). A procedure for doing this was provided by Fletcher and Reeves [6]. It works by constructing two sequences of vectors, starting from an arbitrary initial vector $\mathbf{h}_0 = \mathbf{g}_0$, using the recurrence relations

$$\begin{aligned} \mathbf{g}_{i+1} &= \mathbf{g}_i - \lambda_i \mathbf{A}\cdot\mathbf{h}_i \\ \mathbf{h}_{i+1} &= \mathbf{g}_{i+1} + \gamma_i \mathbf{h}_i \end{aligned} \tag{6.11}$$

where

$$\begin{aligned} \lambda_i &= \frac{\mathbf{g}_i \cdot \mathbf{h}_i}{\mathbf{h}_i \cdot \mathbf{A} \cdot \mathbf{h}_i} \\ \gamma_i &= \frac{\mathbf{g}_{i+1} \cdot \mathbf{g}_{i+1}}{\mathbf{g}_i \cdot \mathbf{g}_i}. \end{aligned} \tag{6.12}$$

For each $i = 0 \ldots n-1$, the vectors constructed in this way have the properties

$$\begin{aligned} \mathbf{h}_i \cdot \mathbf{A} \cdot \mathbf{h}_j &= 0 \\ \mathbf{g}_i \cdot \mathbf{g}_j &= 0 \\ \mathbf{g}_i \cdot \mathbf{h}_j &= 0 \end{aligned}$$

for each $j < i$. In particular, the $\mathbf{h}_i$ are mutually conjugate as we required. These formulae are not supposed to be obvious. The proofs of these statements are out of scope but we might try to look at them when we come to the linear algebra section of the module. For those who are curious see [7]. The result is a very efficient algorithm for finding minima of functions in multiple dimensiona which has been adapted to myriad uses since its discovery.

The"fly in the ointment" in this discussion is the fact that we are assuming throughout that we know the Hessian matrix, $\mathbf{A}$, whereas we actually do not. This is where the real magic comes in: it is possible to contruct the vectors in Eq. (6.11) without knowing $\mathbf{A}$ by exploiting the fact that we have the ability do line minimisations (for example using the golden section search). This is done as follows:

- suppose we are at a point $\mathbf{p}_i$ and set $\mathbf{g}_i = -\nabla f(\mathbf{p}_i)$.
- given a direction $\mathbf{h}_i$ we move along this direction to the line minimum of f in the direction $\mathbf{h}_i$ from $\mathbf{p}_i$ and define $\mathbf{p}_{i+1}$ to be this line minimiser.
- now set $\mathbf{g}_{i+1} = -\nabla f(\mathbf{p}_{i+1})$.
- it turns out that the vector $\mathbf{g}_{i+1}$ constructed in this way is the same as the one obtained from the construction Eqs. (6.11) and (6.12)! Note that the next direction, $\mathbf{h}_{i+1}$, is constructed from $\mathbf{g}_{i+1}$.

To see why this works, we use Eq. (6.8). According to the above prescription

$$\mathbf{g}_i = -\nabla f(\mathbf{p}_i) = \mathbf{b} - \mathbf{A} \cdot \mathbf{p}_i \tag{6.13}$$

$$\mathbf{g}_{i+1} = -\nabla f(\mathbf{p}_{i+1}) = \mathbf{b} - \mathbf{A} \cdot \mathbf{p}_{i+1} \tag{6.14}$$

where $\mathbf{p}_{i+1}$ is the line minimiser of f in the direction $\mathbf{h}_i$ from $\mathbf{p}_i$:

$$\mathbf{p}_{i+1} = \mathbf{p}_i + \lambda_{\min} \mathbf{h}_i, \tag{6.15}$$

for some value of $\lambda_{\min}$ which we will need to find. Substituting Eq. (6.15) into Eq. (6.14) we find

$$\begin{aligned} \mathbf{g}_{i+1} &= \mathbf{b} - \mathbf{A} \cdot (\mathbf{p}_i + \lambda_{\min} \mathbf{h}_i) \\ &= \mathbf{b} - \mathbf{A} \cdot \mathbf{p}_i - \lambda_{\min} \mathbf{A} \cdot \mathbf{h}_i \\ &= \mathbf{g}_i - \lambda_{\min} \mathbf{A} \cdot \mathbf{h}_i. \end{aligned}$$

This is the vector given in Eq. (6.11) provided that $\lambda_{\min}$ has the correct value. To find $\lambda_{\min}$ we note that since $\mathbf{p}_{i+1}$ is a line minimum in the direction $\mathbf{h}_i$ we have

$$0 = \nabla f(\mathbf{p}_{i+1}) \cdot \mathbf{h}_i = \mathbf{g}_{i+1} \cdot \mathbf{h}_i.$$

Substituting our value for $\mathbf{g}_{i+1}$ into this relation we have

$$(\mathbf{g}_i - \lambda_{\min} \mathbf{A} \cdot \mathbf{h}_i) \cdot \mathbf{h}_i = 0 \Rightarrow \lambda_{\min} = \frac{\mathbf{g}_i \cdot \mathbf{h}_i}{\mathbf{h}_i \cdot \mathbf{A} \cdot \mathbf{h}_i}$$

which is exactly what is required by Eq. (6.12). We have therefore constructed the set of directions, $\mathbf{g}_i$ and $\mathbf{h}_i$ required by the Conjugate Gradient Algorithm, Eqs. (6.11) and (6.12), without making reference to the Hessian matrix $\mathbf{A}$!

Bibliography

[1] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Numerical recipes: The art of scientific computing (3rd ed.). Cambridge University Press, New York, NY, USA, 2007.

[2] Nelder-mead method, 2014. http://en.wikipedia.org/wiki/Nelder-Mead_method#mediaviewer/File:Nelder_Mead2.gif.

[3] W. Spendley, G. R. Hext, and F. R. Himsworth. Sequential application of simplex designs in optimisation and evolutionary operation. Technometrics, 4(4):441–461, 1962.

[4] J. A. Nelder and R. Mead. A simplex method for function minimization. The Computer Journal, 7(4):308–313, 1965.

[5] S. Singer and J. Nelder. Nelder-mead algorithm. Scholarpedia, 4(7):2928, 2009. http://www.scholarpedia.org/article/Nelder-Mead_algorithm.

[6] R. Fletcher and C. M. Reeves. Function minimization by conjugate gradients. The Computer Journal, 7(2):149–154, 1964.

[7] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain, 1994. http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf.

Notes 7: Dynamic programming and Dijkstra's algorithm

7.1 Optimal substructure and memoization

We have seen in Sec. 2.1 how the solution of certain problems can be expressed as a combination the solutions of smaller copies of the same problem. This led us to the divide-and-conquer paradigm which allowed us to write down some very succinct algorithms for solving problems like sorting, searching, computing Fourier transforms etc. A problem that can be solved optimally by breaking it into subproblems and then recursively finding the optimal solutions to the subproblems is said to have optimal substructure.

Although divide-and-conquer algorithms are often very concise in terms of coding, for certain types of problems they can be extremely inefficient in terms of run time. This is particularly the case if the recursive subdivision into smaller and smaller problems frequently generates repeated instances of the same problem applied to the same data. When two or more subproblems generated by a recursive subdivision of a larger problem end up applying the same operations to the same input data, the sub-problems are said to be overlapping. Divide-and-conquer algorithms work best when all subproblems are non-overlapping. When a significant number of subproblems are overlapping, it can be more efficient to adopt a strategy in which we solve each subproblem only once, store the result and then look up the answer when the same subproblem recurs later in the calculation. The process of storing results of certain calculations which occur frequently in order to speed up a larger calculation is termed memoization. Memoization is an important tool in the design of algorithms which allows the programmer to trade memory for speed.

To give a concrete example, consider the following recursive Python function to compute the n^{th} Fibonacci number:

```
# Function to compute Fibonacci numbers by naive recursion
def F(n):
   if n == 1 or  n ==0 :
      return n
   else:
      return F(n-1) + F(n-2)
```

At this point, it should be obvious how and why it works. Let us assess the efficiency of this function. Using the same arguments as before, the computational complexity, $F(n)$ satisfies the recursion relation

$$F(n) = F(n-1) + F(n-2)$$

with starting conditions $F(0) = 1$ and $F(1) = 1$. Starting from the ansatz $F(n) = x^n$ and solving the resulting quadratic equation for x, the solution to this recursion relation which satisfies the starting conditions is

$$F(n) = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^{n+1} - \left(\frac{1-\sqrt{5}}{2} \right)^{n+1} \right],$$

which clearly grows exponentially as n increases. At first sight this might seem surprising since the obvious iterative algorithm for computing Fibonacci numbers is clearly $O(n)$. The reason for the poor scaling is that the recursive function $F(n)$ generates an exponentially increasing number of overlapping subproblems as the recursion proceeds. To see this consider the sequence of function

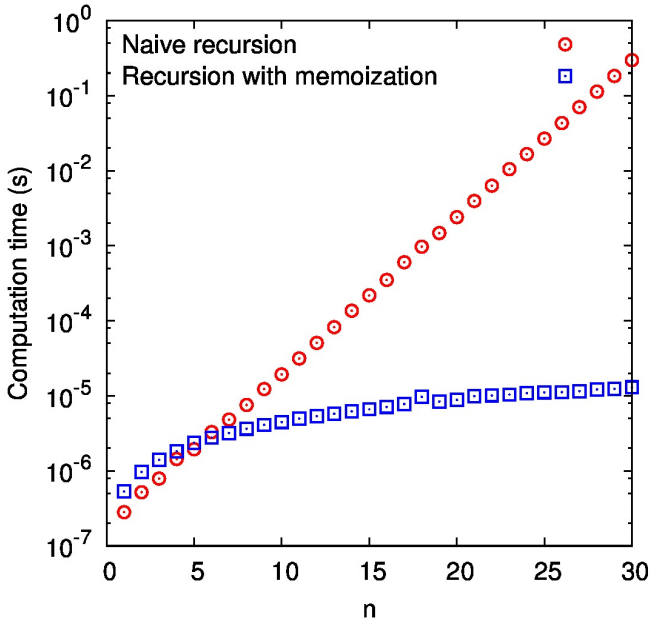


Figure 7.1: Lin-Log plot showing the time required to compute the n^{th} Fibonacci number using the two algorithms described in the text.

calls which are made when computing $F(5)$:

$$\begin{aligned} F(5) &= F(4) + F(3) \\ &= (F(3) + F(2)) + (F(2) + F(1)) \\ &= ((F(2) + F(1)) + (F(1) + F(0))) + ((F(1) + F(0)) + F(1)) \\ &= (((F(1) + F(0)) + F(1)) + (F(1) + F(0))) + ((F(1) + F(0)) + F(1)). \end{aligned}$$

Sifting through this execution stack we see, for example, that $F(2)$ was computed 3 times and $F(3)$ was computed twice. As n grows the duplication of sub-problems becomes sufficiently frequent to result in an exponential time algorithm.

Let us now consider how memoization can be used to greatly speed up this algorithm by storing the results of computations in memory and then looking up the answer when the same computation occurs again later. In order to store the intermediate results, we use a special data structure known as an associative array [1] (called memo in the Python code below). An associative array is a data structure consisting of a set of key-value pairs where each allowed key appears at most once and which supports the operations of addition, removal and modification of pairs in addition to a lookup operation on the keys. In our case the keys will be integers, n, and the corresponding values will be the Fibonacci numbers associated with these integers. Before calling the function, the associative array memo is initialised with the pairs $0:1$ and $1:1$ corresponding to the first two Fibonacci numbers (the base cases in the recursive version above). Here is the code:

```
# Function to compute Fibonacci numbers by recursion with memoization

memo = {0:0, 1:1}

def F2(n):
    if n not in memo :
        memo[n] = F2(n-1) + F2(n-2)
    return memo[n]
```

This is deceptively simple. If the key n is already stored it simply looks up the corresponding value and returns it. If not, it computes it (using the same recursion as before), stores the result for future use and returns the result. The run times of the two versions of the Fibonacci code are plotted as a function of n in Fig. 7.1. The memoization version of the code actually runs in $O(n)$ time. This is because each sub-problem is computed only once.

In reality, one would not use either of these methods to compute Fibonacci numbers. This exercise is purely pedagogical and intended to illustrate the fact that intelligently reusing previously computed results can lead to hug efficiency gains in some problems. This is the feature which generalises far beyond this simple example.

7.2 Dynamic programming

A common class of problems involves making sequences of decisions, each having a known cost or benefit, such that the total cost of the sequence is minimized (or the total benefit is maximised). Such problems often exhibit optimal substructure and are amenable to a solution technique called dynamic programming (DP) which tries to take advantage of overlapping subproblems and solve each subproblem only once.

Consider for example the following toy problem (the checkerboard problem). Consider a checkerboard with $n \times n$ squares and a cost-function $c(i, j)$ which returns a positive cost associated with square i, j (i being the row, j being the column). Here is an example of the 5×5 case. Row and column indices are in bold on the left and bottom rows and costs, $c(i, j)$, are in the interior of the table.

5	2	1	4	10	10
4	3	5	1	4	1
3	2	3	7	3	1
2	4	2	3	2	1
1	0	0	0	0	0
	1	**2**	**3**	**4**	**5**

The checker can start anywhere in row 1 and is only allowed to make the usual moves:

5					
4					
3					
2		X	X	X	
1			O		
	1	**2**	**3**	**4**	**5**

The objective is to find the sequence of moves which minimises the total cost of reaching row n. One strategy would be to choose the move which has the smallest cost at each step. Such a strategy of making locally optimal moves in the hope of finding a global optimum is called a greedy algorithm. Greedy algorithms are usually not optimal but are often used as cheap and easy ways of getting "not-too-bad" solutions. The above table of costs has been designed to set a trap for a greedy strategy to illustrate how locally optimal moves need not necessarily lead to optimal or even near-optimal solutions.

This problem exhibits optimal substructure. That is, the solution to the entire problem relies on solutions to subproblems. Let us define a function $q(i, j)$ as the minimum cost to reach square (i, j). If we can find the values of this function for all the squares at rank n, we pick the minimum and follow that path backwards to get the optimal path. Clearly that $q(i, j)$ is equal to the minimum cost to get to any of the three squares below it plus $c(i, j)$. Mathematically:

$$q(i,j) = \begin{cases} \infty & \text{if } j < 1 \text{ or } j > n \\ 0 & \text{if } j = 1 \\ \min\{q(i-1,j-1), q(i-1,j), q(i-1,j+1)\} + c(i,j) & \text{otherwise} \end{cases} \tag{7.1}$$

The first line is there to make it easier to write the function recursively (no special cases needed for edges):

```
def q(i, j):
    if j < 1 or j > n:
        return infinity
    else if i == 1:
        return c(i, j)
    else:
        return min( q(i-1, j-1), q(i-1, j), q(i-1, j+1) ) + c(i, j)
```

This function will work although, like the Fibonacci example above, it is very slow for large n since recompute the same shortest paths over and over. We can compute it much faster in a bottom-up fashion if we store path-costs in a two-dimensional array q[i, j] rather than using a function thus avoiding recomputation. Before computing the cost of a path, we check the array q[i, j] to see if the path cost is already there. In fact, if you had not learned about recursion, this is probably how you would have instinctively coded up this problem anyway!

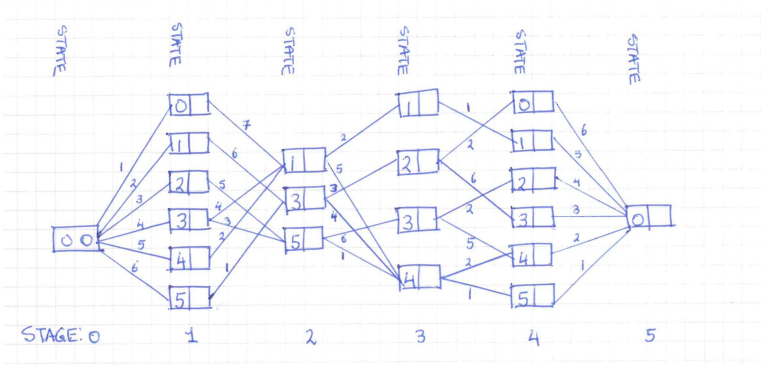
```
def computeMinimumCosts():
   for i from 1 to n
      q[1, i] = 0
   for i from 1 to n
        q[i, 0]     = infinity
        q[i, n + 1] = infinity
   for i from 2 to n
      for j from 1 to n
         m := min(q[i-1, j-1], q[i-1, j], q[i-1, j+1])
         q[i, j] := m + c(i, j)
```

Here are (I think) the contents of the array q[i,j] after running this function:

5	9	6	9	13	13
4	7	9	5	6	3
3	4	7	9	4	2
2	4	2	3	2	1
1	0	0	0	0	0
	1	**2**	**3**	**4**	**5**

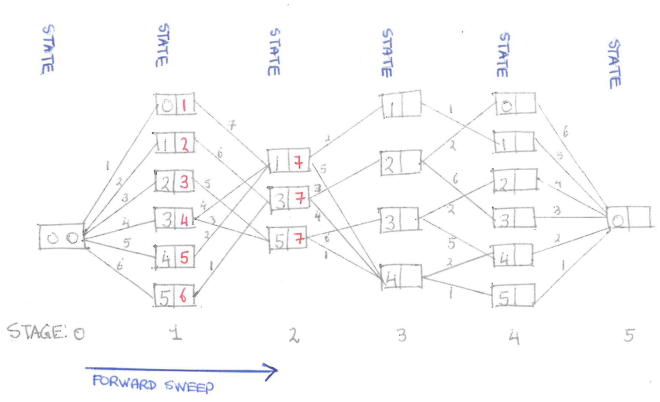
The optimal cost is therefore 6 and the optimal path can be reconstructed by reading backwards through the set of predecessors of the optimal solution. This little problem illustrates the two key features of DP: a forward sweep which stores at each node the cost of the best possible path to reach that node followed by a backtracking stage which follows the precursors of the final optimal state back through the different stages to reconstruct the optimal path. Note that the sequence constructed from DP is globally optimal.

More generally, a problem is amenable to dynamic programming if it can be broken up into a number of discrete stages and within each stage into a number of discrete states. Transitions occur in a directed fashion from states in earlier stages to states in later stages and each transition has an associated cost. The objective is to find the sequence of states going from a particular starting state, i_0, at stage 0 to a particular ending state, i_N, at stage N which minimises the total cost. Such problems are most naturally represented as a weighted directed graph with stages arranged from left to right and edges representing allowed transitions between states in consecutive stages. This is best represented as a picture:

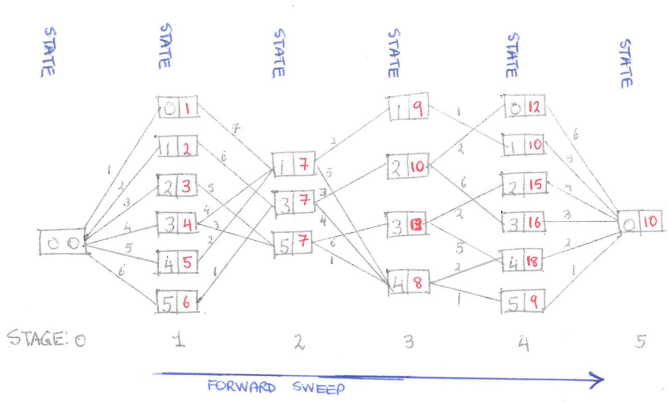


The weight associated with an edge linking state j in stage i to state k in stage $i+1$ is denoted by $w_{jk}(i)$ (just written as integers near the associated edges in the picture). To make this concrete, in the checkerboard example, the stages are the rows, the states are the columns, the edges are the allowed moves from row i to row $i+1$ and the weight of a given edge is the value of $c(i,j)$ on the square at which the move terminates.

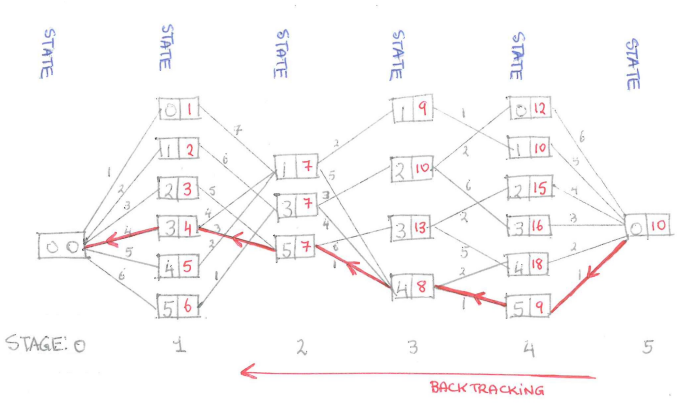
The problem is solved by first making a forward sweep through the graph from left to right. As the sweep proceeds, we label each vertex with a single integer which is the cost of the best way to have reached it:



When the end state is reached, the globally minimal cost becomes known:



A back-tracking pass can then be made, tracing back the set of edges which were followed in order to reach this optimal value. The result is a reconstruction of the optimal path:



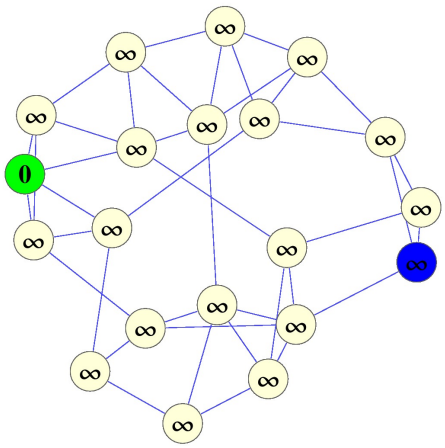
In this case, the optimal path turns out to be $0 \to 3 \to 5 \to 4 \to 5 \to 0$.

7.3 Dijkstra's shortest path algorithm

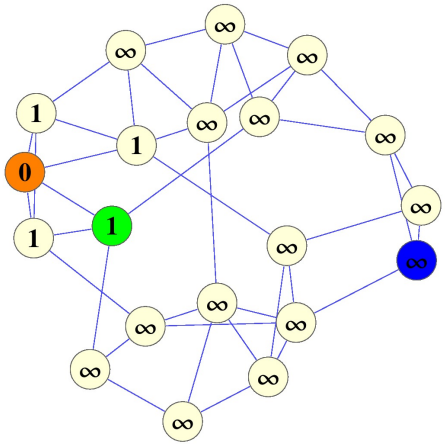
As discussed above it is clear that the problem of finding the shortest path between two nodes on a graph is of fundamental importance to solving DP problems. This is not a problem which is amenable to brute force enumeration since the number of possible paths between two vertices of a graph generally grows exponentially with the number of vertices. There are huge numbers of overlapping subproblems however. This is because if a sequence of vertices $p = \{v_1, v_2, \ldots, v_n\}$ is a shortest path between v_1 and v_n then every contiguous subsequence of vertices of p is itself a shortest path.

Dijkstra's algorithm is the fundamental method of solution for this problem. The idea is to do a sweep through the vertices of the graph, starting at the source node and maintaining a set of labels on each vertex indicating the shortest way to reach it from the source. The algorithm exploits the overlapping subproblems using the fact that if p_1 is a shortest path from v_1 to v_n and p_2 is a shortest path from v_n to v_m then the concatenation of p_2 with p_1 is a shortest path form v_1 to v_m. As a result each vertex is only visited once. We do not assume that the graph is directed or has the stage structure in the DP example above.

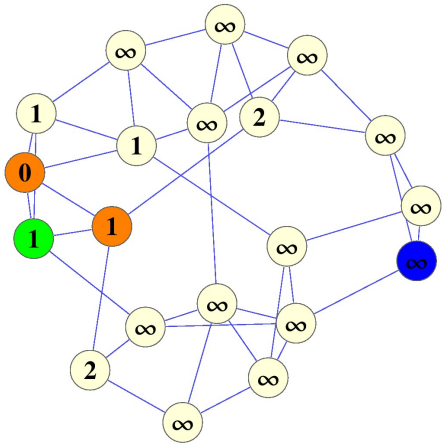
We shall illustrate the procedure in pictures before writing a more formal description of the algorithm. Consider the following graph with 20 nodes:



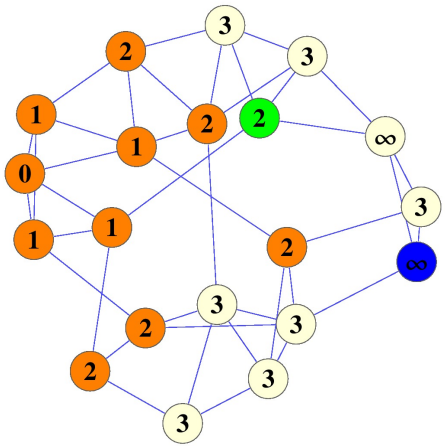
The target node is indicated in blue. All vertices except the source are initially labeled with a distance of ∞ from the source. In this context, ∞ indicates that the distance is not yet known. The source vertex is labeled with a distance of 0. All vertices are initially marked as unvisited (white). To start the search, the current active vertex (indicated in green) is set equal to the source vertex.



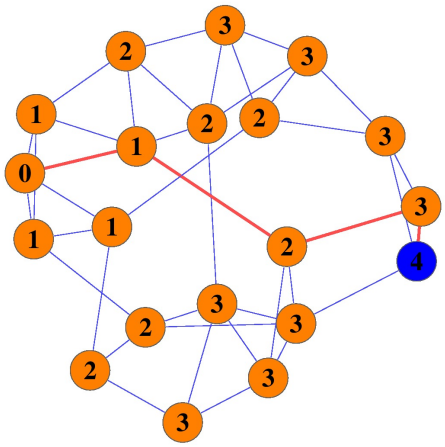
At the first step, the distances of the neighbours of the current active vertex are examined. All of them currently have unknown distance from the source. Since they are neighbours of the current active vertex, there exists a (necessarily shortest) path of length 1 from the current active vertex to each of them and since the current active vertex is a distance 0 from the source, we can set all of distance labels to 1. We then mark the source node as visited (orange) and select the new active vertex to be one of the neighbours of the source. The general rule is to choose the new active vertex to be one of the unvisited vertices whose distance to the source is a minimum.



After two steps, several paths of length 2 have been identified and the active vertex has moved on to the next unvisited vertex at distance 1. The distance labels of the unvisited neighbours of the current active vertex are updated whenever a new path is found which is shorter than the value stored in the current label. The current active vertex is then marked as visited and the new active vertex set equal to one of the unvisited vertices whose distance to the source is a minimum.



After many steps of the procedure, lots of vertices are orange (visited) and labeled with the length of the shortest distance back to the source. The efficiency of the algorithms stems from the fact that it is never necessary to revisit the orange vertices.



In this case, the algorithm is "unlucky": the target vertex is not found until the very last move: 19 intermediate steps were required with the target finally found on the 20th step. Shortest paths like the red one can now be read of by working backwards from the target towards the source. From any visited vertex it is always possible to find a neighbour which is one step nearer to the source. There may be several such neighbours however. In this example, there are three neighbours of the target at distance 3, each of which leads to a valid shortest path.

Here is some pseudocode for a simple version of Dijkstra's algorithm which finds the length of the shortest path between a source and target vertex in a graph G. It is assumed that G is fully connected so that at least one shortest path exists between any two vertices.

```
input : Graph G
input : vertex source
input : vertex target
// Initialisations
dist [source ] = 0 ;                          // Source is distance 0 from itself
for vertex v ∈ G do
    if v! = source then
        dist[v] = ∞;                                  // ∞ here means unknown
    end
    add v to unvisitedQ ;                  // maintain list of unvisited vertices
end
v = source ;                          // Set current active vertex to the source
dv = 0 ;                                              // Initial distance is zero
// Main loop starts here
while v! = target do
    foreach neighbour u of v ;            // loop over neighbours of active vertex
     do
        if u ∈ unvisitedQ ;                    // only check those not yet visited
         then
            du = dist [u] ;                   // current estimate of distance of u
            if dv + 1 < du ;             // check if path to u through v is shorter
             then
                dist [u] = dv + 1 ;     // if yes, update estimate of distance of u
            end
        end
    end
    remove v from unvisitedQ ;                      // v will never be visited again
    v = fetch min distance vertex from unvisitedQ ;   // set new active vertex to be
     an
    ;                                               // unvisited vertex with minimum
    ;                                                    // distance from the source
    dv = dist [v]
end
```

A Python implementation of the algorithm is available in the class www site. Some comments on the above pseudocode:

- In this implementation of the algorithm, all edges have length 1. It is easy to generalise to the case where the edges have other weights by replacing line 18 with
  dist [u] = dv + **length** (u,v)
  where length(v_1,v_2) is the function which gives the weight of the edge linking vertices v_1 and v_2.
- Line 23 assumes that a function has been provided which searches the list of unvisiting vertices and associated list of distances and returns an unvisited vertex whose distance to the source is minimal. In the sample code, this is done using simple linear search. For this implementation the typical runtime of the algorithm (at least for sparse graphs) is $O(|V|^2)$ where $|V|$ is the number of vertices. This is because roughly we do one loop over the $|V|$ vertices, each of which requires a linear search of an array of length $|V|$. If the number of vertices is large, linear search maybe start to slow the algorithm down significantly. This can be remedied by using some of the more sophisticated searching and sorting procedures which we discussed earlier. A specialised data structure called a priority queue [2] supports insertion, deletion and retrieval of minimal key-value pairs (in this case the key would be the distance and the value would be the vertex) in $O(\log n)$ time where n is the number of entries in the queue.
- As written, the algorithm only calculates the length of the shortest path from source to target. In dynamic programming terms, it does the "sweep" stage of the DP solution. If we wish to obtain an actual path then we need to do the backtracking stage too. This means starting at the

target, stepping to a neighbouring vertex whose distance to the source is minimal and iterating this procedure until the source is reached. Here is some pseudocode which achieves this:

```
v = target ;                          // Set current active vertex to the target
dv = dist [v] ;                       // Initial distance is shortest path length
spath = [v]
while v! = source do
    foreach neighbour u of v ;        // loop over neighbours of active vertex
     do
        if u ∉ unvisitedQ ;           // only check those visited during sweep stage
         then
            du = dist [u] ;                        // distance of u from source
            if du == dv -1 ;          // check if u is a step closer to the source
             then
                v=u ;                             // if yes, update active vertex
            end
        end
    end
    dv = dist [v];                    // Update distance of new active vertex
    add v to spath ;                  // Add new active vertex to the path
end
return spath
```

As we have remarked before, there may be more than one shortest path. This procedure will only find one of them but can be adapted to find all of them.

- If instead of terminating when the target vertex is found, line 10 were changed to terminate when the list of unvisited vertices is empty, then the sweep stage will have found the length of the shortest path from the source to any other vertex on the graph. Therefore in terms of computational complexity, finding all shortest paths is as easy (or as hard) as finding one shortest path since both are $O(|V|^2)$ operations!

Bibliography

[1] Associative array, 2014. http://en.wikipedia.org/wiki/Associative_array.

[2] Priority queue, 2014. http://en.wikipedia.org/wiki/Priority_queue.

Notes 8: Solving ODEs and Fourier transforms

8.1 Ordinary differential equations and dynamical systems

Differential equations are one of the basic tools of mathematical modelling and no discussion of computational methods would be complete without a discussion of methods for their numerical solution.. An ordinary differential equation (ODE) is an equation involving one or more derivatives of a function of a single variable. The word "ordinary" serves to distinguish it from a partial differential equation (PDE) which involves one or more partial derivatives of a function of several variables. Let u be a real–valued function of a single real variable, t, and F be a real–valued function of t, $u(t)$ and its first $m-1$ derivatives. The equation

$$\frac{d^m u}{dt^m}(t) = F\left(t, u(t), \frac{du}{dt}(t), \ldots, \frac{d^{m-1}u}{dt^{m-1}}(t)\right) \tag{8.1}$$

is an ordinary differential equation of order m. Equations for which F is has no explicit dependence on t are referred to as autonomous. The general solution of an m^{th} order ODE involves m constants of integration. We frequently encounter three types of problems involving differential equations:

- **Initial Value Problems (IVPs)**
  The value of $u(t)$ and a sufficient number of derivatives are specified at an initial time, $t = 0$ and we require the values of $u(t)$ for $t > 0$.
- **Boundary Value Problems (BVPs)**
  Boundary value problems arise when information on the behaviour of the solution is specified at two different points, x_1 and x_2, and we require the solution, $u(x)$, on the interval $[x_1, x_2]$. Choosing to call the dependent variable x instead of t hints at the fact that boundary value problems frequently arise in modelling spatial behaviour rather than temporal behaviour.
- **Eigenvalue Problems (EVPs)**
  Eigenvalue problems are boundary value problems involving a free parameter, λ. The objective is to find the value(s) of λ, the eigenvalue(s), for which the problem has a solution and to determine the corresponding eigenfunction(s), $u_\lambda(x)$.

We will focus exclusively on IVPs here but much of what we learn is relevant for BVPs and EVPs.

Systems of coupled ODEs arise very often. They can arise as a direct output of the modelling or as a result of discretising the spatial part of a PDE. Many numerical approaches to solving PDEs involve approximating the PDE by a (usually very large) system of ODEs. In practice, we only need to deal with systems of ODEs of first order. This is because an ODE of order m can be rewritten as a system of first order ODEs of dimension m. To do this for the m^{th} order ODE Eq.(8.1), define a vector $\mathbf{u}(t)$ in $\mathbb{R}^m$ whose components [1], $u^{(i)}(t)$, consist of the function $u(t)$ and its first $m-1$ derivatives:

$$\mathbf{u}(t) = \left(u^{(0)}(t), \ldots u^{(m-1)}(t)\right) = \left(u(t), \frac{du}{dt}(t), \ldots, \frac{d^{m-1}u}{dt^{m-1}}(t)\right).$$

From this point on, we will not write the explicit t-dependence of $\mathbf{u}(t)$ and its components unless it is necessary for clarity. We define another vector, $\mathbf{F} \in \mathbb{R}^m$, whose components will give the right hand sides:

$$\mathbf{F}(t, \mathbf{u}) = \left(u^{(1)}, \ldots, u^{(m-1)}, F(t, \mathbf{u})\right).$$

By construction, Eq.(8.1) is equivalent to the first order system of ODEs

$$\frac{d\mathbf{u}}{dt} = \mathbf{F}(t, \mathbf{u}). \tag{8.2}$$

[1] The clumsy superscript notation, $u^{(i)}$ for the components of $\mathbf{u}$ is adopted to accommodate the subsequent use of the notation u_i to denote the value of a function $u(t)$ at a discrete time point, t_i.

The initial conditions given for Eq. (8.1) can be assembled together to provide an initial condition $\mathbf{u}(0) = \mathbf{U}$. We can also do away with the distinction between autonomous and non-autonomous equations be introducing an extra component, $u^{(m)}$, of $\mathbf{u}$ which satisfies the trivial ODE

$$\frac{du^{(m)}}{dt} = 1 \qquad \text{with } u^{(m)}(0) = 0. \tag{8.3}$$

For this reason, we will focus primarily on developing numerical algorithms for autonomous first order systems of equations in dimension m

$$\frac{d\mathbf{u}}{dt} = \mathbf{F}(\mathbf{u}) \qquad \text{with } \mathbf{u}(0) = \mathbf{U}. \tag{8.4}$$

8.2 Timestepping and simple Euler methods

8.2.1 Timestepping and the Forward Euler Method

In the computer, we approximate $\mathbf{u}$ by its values at a finite number of points. The process of replacing a function by an array of values at particular points is called discretisation. We replace $\mathbf{u}(t)$ with $t \in [t_1, t_2]$ by a list of values, $\{\mathbf{u}_i \ : \ i = 0 \ldots N\}$ where $\mathbf{u}_i = \mathbf{u}(t_i)$, $t_i = t_1 + ih$ and $h = (t_2 - t_1)/N$. There is no reason why the discretisation should be on a set of uniformly spaced points but we will initially consider this case. We have already seen in Sec. 3.2.1 how knowing the values of a function at a stencil of discrete points allows us to approximate the derivative of the function. Furthermore we knew the error in the approximation using Taylor's Theorem. We can turn this reasoning around: if we know $\mathbf{u}_i$ and $\frac{d\mathbf{u}}{dt}(t_i)$ then we can approximate $\mathbf{u}_{i+1}$ (of course since $\mathbf{u}(t)$ is now a vector-valued function we must use the multivariate version of Taylor's theorem):

$$\mathbf{u}_{i+1} = \mathbf{u}_i + h\,\frac{d\mathbf{u}}{dt}(t_i) + O(h^2). \tag{8.5}$$

For the system of ODEs (8.4) this gives:

$$\mathbf{u}_{i+1} = \mathbf{u}_i + h\,\mathbf{F}_i + O(h^2), \tag{8.6}$$

where we have adopted the obvious notation $\mathbf{F}_i = \mathbf{F}(\mathbf{u}(t_i))$. Starting from the initial condition for Eq. (8.4) and knowledge of $\mathbf{F}(\mathbf{u})$ we obtain an approximate value for $\mathbf{u}$ a time h later. We can then iterate the process to obtain an approximation to the values of $\mathbf{u}$ at all subsequent times, t_i. While this iteration might be possible to carry out analytically in some simple cases (for which we are likely to be able to solve the original equation anyway), in general it is ideal for computer implementation. This iteration process is called time-stepping. Eq. (8.6) provides the simplest possible time-stepping method. It is called the Forward Euler Method.

8.2.2 Stepwise vs Global Error

In Eq. (8.6), we refer to $O(h^2)$ as the stepwise error. This is distinct from the global error, which is the total error which occurs in the approximation of $\mathbf{u}(t_2)$ if we integrate the equation from t_1 to t_2 using a particular timestepping algorithm. If we divide the time domain into N intervals of length $h = (t_2 - t_1)/N$ then $N = (t_2 - t_1)/h$. If we make a stepwise error of $O(h^n)$ in our integration routine, then the global error therefore $O((t_2 - t_1)h^{n-1})$. Hence the global error for the Forward Euler Method os $O(h)$. This is very poor accuracy. This is one reason why the Forward Euler Method is almost never used in practice.

8.2.3 Backward Euler Method - implicit vs explicit timestepping

We could equally well have used the backward difference formula, Eq. (3.9), to derive a time-stepping algorithm in which case we would have found

$$\mathbf{u}_{i+1} = \mathbf{u}_i + h\,\mathbf{F}_{i+1} + O(h^2). \tag{8.7}$$

Now, $\mathbf{u}_{i+1}$ (the quantity we are trying to find) enters on both sides of the equation. This means we are not, in general in a position to write down an explicit formula for $\mathbf{u}_{i+1}$ in terms of $\mathbf{u}_i$ as we did for

the Forward Euler Method, Eq. (8.6). Rather we are required to solve a (generally nonlinear) system of equations to find the value of $\mathbf{u}$ at the next timestep. For non-trivial $\mathbf{F}$ this may be quite hard since, as we learned in Sec. 4.2, root finding in higher dimensions can be a difficult task.

The examples of the Forward and Backward Euler methods illustrates a very important distinction between different timestepping algorithms: explicit vs implicit. Forward Euler is an explicit method. Backward Euler is implicit. Although they both have the same degree of accuracy - a stepwise error of $O(h^2)$ - the implicit version of the algorithm typically requires a lot more work per timestep. However it has superior stability properties.

8.3 Predictor-Corrector methods

8.3.1 Implicit Trapezoidal method

We arrived at the simple Euler methods by approximating $\mathbf{u}$ by its Taylor series. A complementary way to think about it is to begin from the formal solution of Eq. (8.4):

$$\mathbf{u}_{i+1} = \mathbf{u}_i + \int_{t_i}^{t_i+h} \mathbf{F}(\mathbf{u}(\tau))\, d\tau, \tag{8.8}$$

and think of how to approximate the integral,

$$I = \int_{t_i}^{t_i+h} \mathbf{F}(\mathbf{u}(\tau))\, d\tau. \tag{8.9}$$

The simplest possibility is to use the rectangular approximations familiar from the definition of the Riemann integral. We can use either a left or right Riemann approximation:

$$I \approx h\mathbf{F}(\mathbf{u}(t_i)) \tag{8.10}$$

$$I \approx h\mathbf{F}(\mathbf{u}(t_{i+1})). \tag{8.11}$$

These approximations obviously give us back the Forward and Backward Euler Methods which we have already seen. A better approximation would be to use the Trapezoidal Rule, Eq. (3.12) :

$$I \approx \frac{1}{2}h\left[\mathbf{F}(\mathbf{u}(t_i)) + \mathbf{F}(\mathbf{u}(t_{i+1}))\right].$$

Eq. (8.8) then gives the timestepping rule

$$\mathbf{u}_{i+1} = \mathbf{u}_i + \frac{h}{2}(\mathbf{F}_i + \mathbf{F}_{i+1}). \tag{8.12}$$

This is known as the Implicit Trapezoidal Method. The stepwise error in this approximation is $O(h^3)$. To see this, we Taylor expand the terms sitting at later times in Eq. (8.12) and compare the resulting expansion to the true Taylor expansion. It is worth writing this example out in some detail since it is a standard way of deriving the stepwise error for any timestepping algorithm. For simplicity, let us assume that we have a scalar equation,

$$\frac{du}{dt} = F(u). \tag{8.13}$$

The extension of the analysis to the vector case is straight-forward but indicially messy. The true solution at time t_{i+1} up to order h^3 is, from Taylor's Theorem and the Chain Rule:

$$\begin{aligned} u_{i+1} &= u_i + h\frac{du}{dt}(t_i) + \frac{h^2}{2}\frac{d^2u}{dt^2}(t_i) + O(h^3) \\ &= u_i + hF_i + \frac{1}{2}h^2 F_i\, F_i' + O(h^3) \end{aligned}$$

Note that we have used Eq. (8.13) and the chain rule to replace the second derivative of u wrt t at time t_i by $F_i\, F_i'$. Make sure you understand how this works. Let us denote our approximate solution by $\widetilde{u}(t)$. From Eq. (8.12)

$$\widetilde{u}_{i+1} = u_i + \frac{h}{2}\left[F(u_i) + F(u(t_{i+1}))\right]$$

We can write

$$\begin{aligned} F(u(t_{i+1})) &= F(u(t_i)) + h\frac{dF}{dt}(u(t_i)) + O(h^2) \\ &= F_i + hF_i'F_i + O(h^2). \end{aligned}$$

Substituing this back gives

$$\widetilde{u}_{i+1} = u_i + hF_i + \frac{1}{2}h^2 F_i\, F_i' + O(h^3).$$

We now see that the difference between the true value of u_{i+1} and the approximate value given by Eq. (8.12) is $\widetilde{u}_{i+1} - u_{i+1} = O(h^3)$. Hence the Implicit Trapezoidal Method has an $O(h^3)$ stepwise error. Strictly speaking we have only shown that the stepwise error is at most $O(h^3)$. We have not computed the $O(h^3)$ terms explicitly to show that they do not in general cancel as the $O(h^2)$ ones do. Take it on faith that they do not (or do the calculation yourself in an idle moment).

8.3.2 Improved Euler method

The principal drawback of the Implicit Trapezoidal Method is that it is implicit and hence computationally expensive and tricky to code (but not for us now that we know how to find roots of nonlinear equations!). Can we get higher order accuracy with an explicit scheme? A way to get around the necessity of solving implicit equations is try to "guess" the value of $\mathbf{u}_{i+1}$ and hence estimate the value of $\mathbf{u}_{i+1}$ to go into the RHS of Eq. (8.12). How do we "guess"? One way is to use a less accurate explicit method to predict $\mathbf{u}_{i+1}$ and then use a higher order method such as Eq. (8.12) to correct this prediction. Such an approach is called a Predictor–Corrector Method. Here is the simplest example:

- Step 1
  Make a prediction, $\mathbf{u}^*_{i+1}$ for the value of $\mathbf{u}_{i+1}$ using the Forward Euler Method:

  $$\mathbf{u}^*_{i+1} = \mathbf{u}_i + h\mathbf{F}_i,$$

  and calculate

  $$\mathbf{F}^*_{i+1} = \mathbf{F}(\mathbf{u}^*_{i+1}).$$

- Step 2
  Use the Trapezoidal Method to correct this guess:

  $$\mathbf{u}_{i+1} = \mathbf{u}_i + \frac{h}{2}\left[\mathbf{F}_i + \mathbf{F}^*_{i+1}\right]. \tag{8.14}$$

Eq. (8.14) is called the Improved Euler Method. It is explicit and has a stepwise error of $O(h^3)$. This can be shown using an analysis similar to that done above for the Implicit Trapezoidal Method. The price to be paid is that we now have to evaluate $\mathbf{F}$ twice per timestep. If $\mathbf{F}$ is a very complicated function, this might be worth considering but typilcally the improved accuracy more than compensates for the increased number of function evaluations.

There are two common approaches to making further improvements to the error:

1. Use more points to better approximate the integral, Eq. (8.9). This leads to class of algorithms known as multistep methods. These have the disadvantage of needing to remember multiple previous values (a problem at $t = 0$!) and will not be discussed here although you can read about them here [1] or [2, Chap. 17].
2. Use predictors of the solution at several points in the interval $[t_i, t_{i+1}]$ and combine them in clever ways to cancel errors. This approach leads to a class of algorithms known as Runge-Kutta methods.

8.4 Using adaptive timestepping to deal with multiple time scales and singularities

Up until now we have characterised timestepping algorithms as $h \to 0$. In practice we need to operate at a finite value of h. How do we choose it? Ideally we would like to choose the timestep such that the error per timestep is less than some threshold, ϵ. We measure the error by comparing

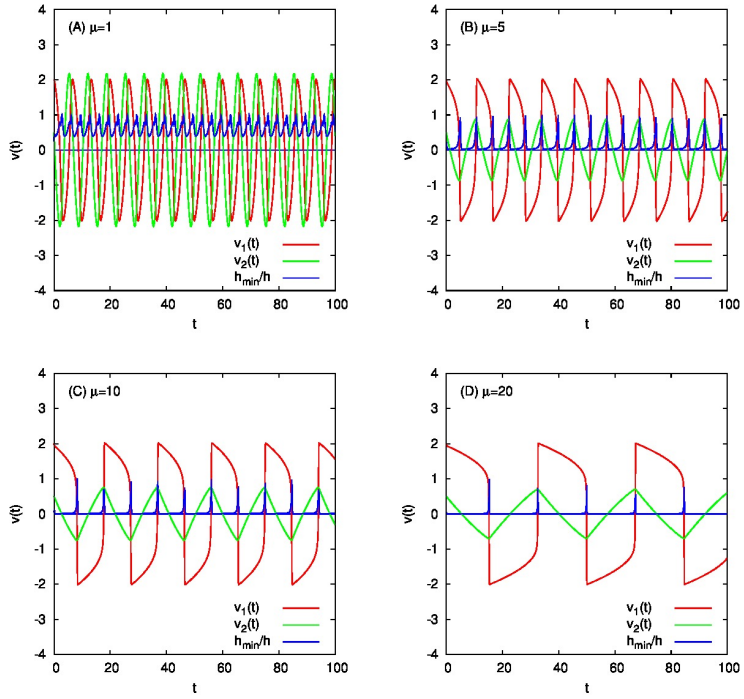


Figure 8.1: Solutions of Eq. (8.15) for several different values of μ obtained using the forward Euler Method with adaptive timestepping. Also shown is how the timestep varies relative to its global minimum value as the solution evolves.

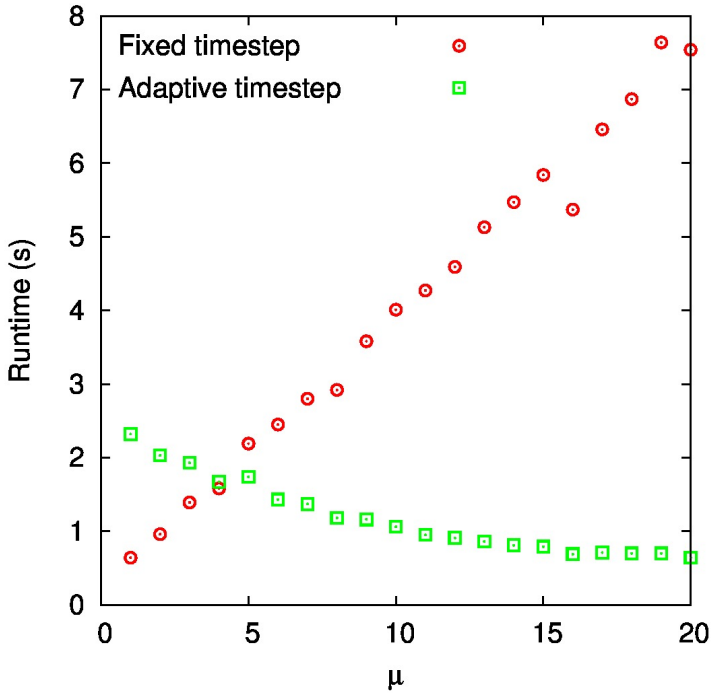


Figure 8.2: Comparison of the performance of the simple Euler method with adaptive and fixed timestepping strategies applied to Eq. (8.15) for several different values of μ.

the numerical solution at a grid point, $\widetilde{\mathbf{u}}_i$, to the exact solution, $\mathbf{u}(t_i)$, assuming it is known. Two criteria are commonly used:

$$\begin{aligned} E_a(h) = |\widetilde{\mathbf{u}}_i - \mathbf{u}_i| \leq \epsilon & \quad \text{Absolute error threshold,} \\ E_r(h) = \frac{|\widetilde{\mathbf{u}}_i - \mathbf{u}_i|}{|\mathbf{u}_i|} \leq \epsilon & \quad \text{Relative error threshold.} \end{aligned}$$

Intuitively (and mathematically from Taylor's Theorem) the error is largest when the solution is rapidly varying. Often the solution does not vary rapidly at all times. By setting the timestep in this situation so that the error threshold is satisfied during the intervals of rapid variation, we end up working very inefficiently during the intervals of slower variation where a much larger timestep would have easily satisfied the error threshold. Here is a two–dimensional example which you already know: a relaxation oscillator when $\mu \gg 1$:

$$\begin{aligned} \frac{dx}{dt} &= \mu(y - (\frac{1}{3}x^3 - x)) \\ \frac{dy}{dt} &= -\frac{1}{\mu}x. \end{aligned} \tag{8.15}$$

The solution of this system has two widely separated timescales - jumps which proceed on a time of $O(\frac{1}{\mu})$ (very fast when $\mu \gg 1$) and crawls which proceed on a timescale of μ (very fast when $\mu \gg 1$). To integrate Eqs. (8.15) efficiently for a given error threshold, we need to take small steps during the jumps and large ones during the crawls. This process is called adaptive timestepping.

The problem, of course, is that we don't know the local error for a given timestep since we do not know the exact solution. So how do we adjust the timstep if we do not know the error? A common approach is to use trial steps: at time i we calculate one trial step starting from the current solution, $\mathbf{u}_i$ using the current value of h and obtain an estimate of the solution at the next time which we shall call $\mathbf{u}^{\mathrm{B}}_{i+1}$. We then calculate a second trial step starting from $\mathbf{u}_i$ and taking two steps, each of length $h/2$, to obtain a second estimate of the solution at the next time which we shall call $\mathbf{u}^{\mathrm{S}}_{i+1}$. We can then estimate the local error as

$$\Delta = \left|\mathbf{u}^{\mathrm{B}}_{i+1} - \mathbf{u}^{\mathrm{S}}_{i+1}\right|.$$

We can monitor the value of Δ to ensure that it stays below the error threshold.

If we are using an n^{th} order method, we know that $\Delta = ch^n$ for some c. The most efficient choice of step is that for which $\Delta = \epsilon$ (remember ϵ is our error threshold. Thus we would like to choose the new timestep, $\tilde{h}$ such that $c\tilde{h}^n = \epsilon$. But we know from the trial step that $c = \Delta/h^n$. From this we can obtain the following rule to get the new timestep from the current step, the required error threshold and the local error estimated from the trial steps:

$$\tilde{h} = \left(\frac{\epsilon}{\Delta}\right)^{\frac{1}{n}} h. \tag{8.16}$$

It is common to include a "safety factor", $\sigma_1 < 1$ to ensure that we stay a little below the error threshold:

$$\tilde{h}_1 = \sigma_1 \left(\frac{\epsilon}{\Delta}\right)^{\frac{1}{n}} h. \tag{8.17}$$

Equally, since the solutions of ODE's are usually smooth, it is often sensible to ensure that the timestep does not increase or decrease by more than a factor of σ_2 (2 say) in a single step. To do this we impose the second constraint:

$$\begin{aligned} \tilde{h} &= \max\left\{\tilde{h}_1, \frac{h}{\sigma_2}\right\} \\ \tilde{h} &= \min\left\{\tilde{h}_1, \sigma_2 h\right\}. \end{aligned}$$

The improvement in performance obtained by using adaptive timestepping to integrate Eq. (8.15) with the simple forward Euler method is plotted as a function of the parameter μ in Fig. (8.2).

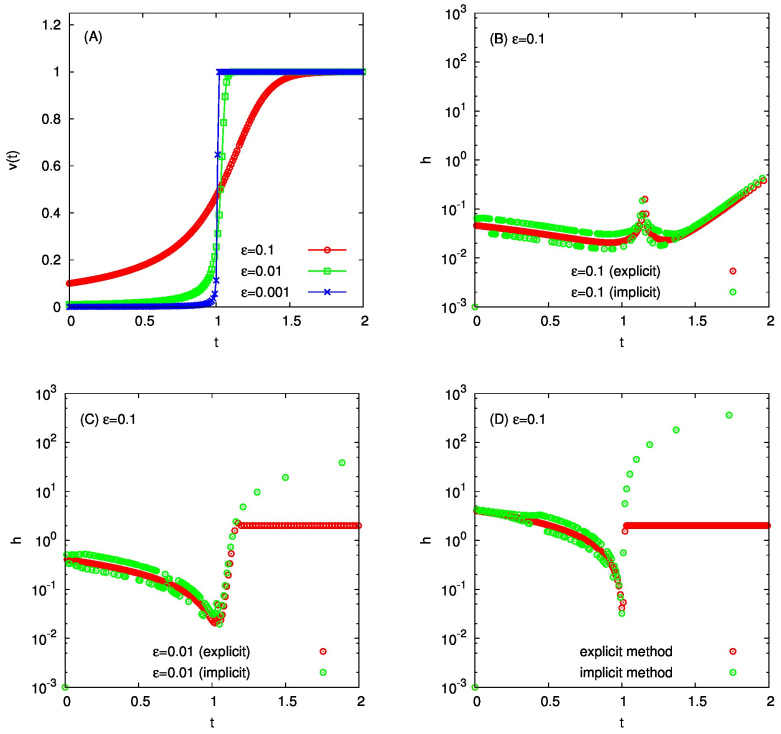


Figure 8.3: Comparison between behaviour of the forward and backward Euler schemes with adaptive timestepping applied to the problem in Eq. (8.19) for a range of values of ϵ.

Another situation in which adaptive timestepping is essential is when the ODE which we are trying to solve has a singularity. A singularity occurs at some time, $t = t^*$ if the solution of an ODE tends to infinity as $t \to t^*$. Naturally this will cause problems for any numerical integration routine. A properly functioning adaptive stepping strategy should reduce the timestep to zero as a singularity is approached. As a simple example, consider the equation,

$$\frac{du}{dt} = \lambda u^2.$$

with initial data $u(0) = u_0$. This equation is separable and has solution

$$u(t) = \frac{u_0}{1 - \lambda\, u_0\, t}.$$

This clearly has a singularity at $t^* = (\lambda\, u_0)^{-1}$.

8.5 Stiffness and implicit methods

Stiffness is an unpleasant property of certain problems which causes adaptive algorithms to select very small timesteps even though the solution is not changing very quickly. There is no uniformly accepted definition of stiffness. Whether a problem is stiff or not depends on the equation, the initial condition, the numerical method being used and the interval of integration. The common feature of stiff problems elucidated in a nice article by Moler [3], is that the required solution is slowly varying but "nearby" solutions vary rapidly. The archetypal stiff problem is the decay equation,

$$\frac{du}{dt} = -\lambda\, u \tag{8.18}$$

with $\lambda >> 1$. Efficient solution of stiff problems typically requires the use of implicit algorithms. Explicit algorithms with proper adaptive stepping will work but usually take an unfeasibly long time. Here is a more interesting nonlinear example taken from [3]:

$$\frac{dv}{dt} = v^2 - v^3. \tag{8.19}$$

with initial data $v(0) = \epsilon$. Find the solution on the time interval $[0, 2/\epsilon]$. Its solution is shown in Fig. 8.3(A). The problem becomes stiff as ϵ is decreased. We see, for a given error tolerance (in this case, a relative error threshold of 10^{-5}), that if $\epsilon \ll 1$ the implicit backward Euler scheme can compute the latter half of the solution (the stiff part) with enormously larger timesteps than the corresponding explicit scheme. An illuminating animation and further discussion of stiffness with references is available on Scholarpedia [4].

8.6 Runge-Kutta methods

We finish our discussion of methods for integration of ODEs with a brief discussion of the Runge-Kutta methods since these are the real workhorses of the business. When you call a black box integration routine in MatLab or Mathematica to integrate an ODE, odds are it is using a Runge-Kutta method so it is worth knowing a little about how they work. Runge-Kutta methods aim to retain the accuracy of the multistep predictor corrector methods but without having to use more than the current value of $\mathbf{u}$ to predict the next one - ie they are self-starting. The idea is roughly to make several predictions of the value of the solution at several points in the interval $[t_1, t_{i+1}]$ and then weight them cleverly so as to cancel errors.

An n^{th} order Runge–Kutta scheme for Eq. (8.4) looks as follows. We first calculate n estimated values of $\mathbf{F}$ which are somewhat like the predictors used in Sec. 8.3:

$$\begin{aligned}
\mathbf{f}_1 &= \mathbf{F}(\mathbf{u}_i) \\
\mathbf{f}_2 &= \mathbf{F}(\mathbf{u}_i + a_{2\,1}\, h\, \mathbf{f}_1) \\
\mathbf{f}_3 &= \mathbf{F}(\mathbf{u}_i + a_{3\,1}\, h\, \mathbf{f}_1 + a_{3\,2}\, h, \mathbf{f}_2) \\
&\vdots \\
\mathbf{f}_n &= \mathbf{F}(\mathbf{u}_i + a_{n\,1}\, h\, \mathbf{f}_1 + \ldots + a_{n\,n-1}\, h, \mathbf{f}_{n-1})
\end{aligned}$$

We then calculate $\mathbf{u}_{i+1}$ as

$$\mathbf{u}_{i+1} = \mathbf{u}_i + h\,(b_1\mathbf{f}_1 + b_2\mathbf{f}_2 + \ldots + b_n\mathbf{f}_n). \tag{8.20}$$

The art lies in choosing the $a_{i\,j}$ and b_i such that Eq. (8.20) has a stepwise error of $O(h^{n+1})$. The way to do this is by comparing Taylor expansions as we did to determine the accuracy of the Improved Euler Method in Sec. 8.3 and choosing the values of the constants such that the requisite error terms vanish. It turns out that this choice is not unique so that there are actually parametric families of Runge-Kutta methods of a given order.

We shall derive a second order method explicitly for a scalar equation, EQ. (8.13). Derivations of higher order schemes provide nothing new conceptually but require a lot more algebra. Extension to the vector case is again straightforward but requires care with indices. Let us denote our general 2-stage Runge-Kutta algorithm by

$$\begin{aligned}
f_1 &= F(u_i) \\
f_2 &= F(u_i + a\, h\, f_1) \\
u_{i+1} &= u_i + h\,(b_1 f_1 + b_2 f_2).
\end{aligned}$$

Taylor expand f_2 up to second order in h:

$$\begin{aligned}
f_2 &= F(u_i) + a\, h\, F_i\, \frac{dF}{du}(u_i) + O(h^2) \\
&= F_i + a\, h\, F_i\, F_i' + O(h^2)
\end{aligned}$$

Our approximate value of u_{i+1} is then

$$\widetilde{u}_{i+1} = u_i + (b_1 + b_2)hF_i + ab_2h^2F_i\,F_i' + O(h^3).$$

If we compare this to the usual Taylor expansion of u_{i+1} we see that we can make the two expansions identical up to $O(h^3)$ if we choose

$$\begin{aligned}
b_1 + b_2 &= 1 \\
ab_2 &= \frac{1}{2}.
\end{aligned}$$

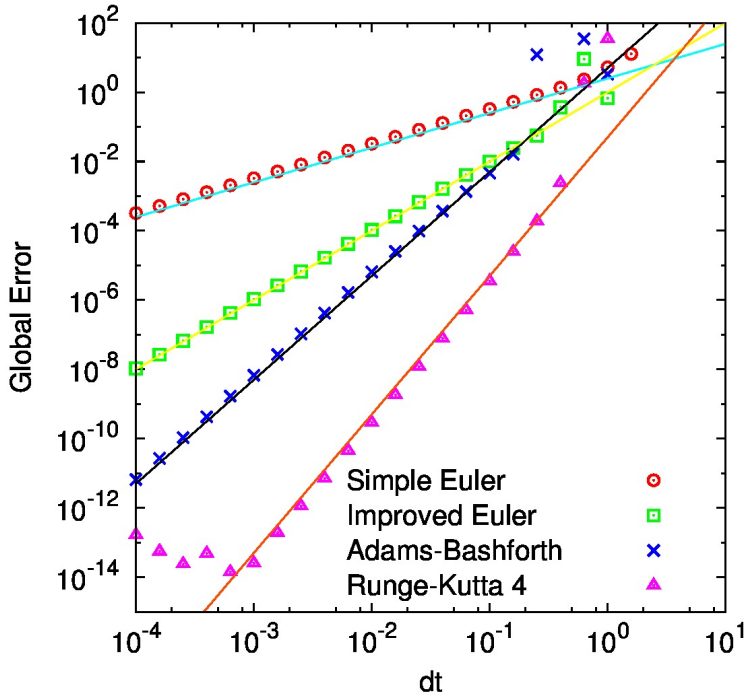


Figure 8.4: Log-Log plot showing the behaviour of the global error as a function of step size for several different timestepping algorithms applied to Eq. (8.23). The straight lines show the theoretically expected error, h^n, where $n = 1$ for the simple Euler method, $n = 2$ for the improved Euler method, $n = 3$ for the Adams–Bashforth method and $n = 4$ for the 4th order Runge–Kutta method.

A popular choice is $b_1 = b_2 = \frac{1}{2}$ and $a = 1$. This gives, what is often considered the "standard" second order Runge-Kutta scheme:

$$\begin{aligned} f_1 &= F(u_i) \\ f_2 &= F(u_i + h\, f_1) \\ u_{i+1} &= u_i + \frac{h}{2}\,(f_1 + f_2). \end{aligned} \tag{8.21}$$

Perhaps it looks familiar. The standard fourth order Runge–Kutta scheme, with a stepwise error of $O(h^5)$, is really the workhorse of numerical integration since it has a very favourable balance of accuracy, stability and efficiency properties. It is often the standard choice. We shall not derive it but you would be well advised to use it in your day-to-day life. It takes the following form:

$$\begin{aligned} \mathbf{f}_1 &= \mathbf{F}(\mathbf{u}_i) \\ \mathbf{f}_2 &= \mathbf{F}(\mathbf{u}_i + \frac{h}{2}\,\mathbf{f}_1) \\ \mathbf{f}_3 &= \mathbf{F}(\mathbf{u}_i + \frac{h}{2}\,\mathbf{f}_2) \\ \mathbf{f}_4 &= \mathbf{F}(\mathbf{u}_i + h\,\mathbf{f}_3) \\ \mathbf{u}_{i+1} &= \mathbf{u}_i + \frac{h}{6}(\mathbf{f}_1 + 2\mathbf{f}_2 + 2\mathbf{f}_3 + \mathbf{f}_4). \end{aligned} \tag{8.22}$$

Fig. 8.4 compares the error in several of the integration routines discussed above when applied to the test problem

$$\frac{d^2v}{dt^2} + 2\,t\frac{dv}{dt} - v = 0, \tag{8.23}$$

with initial conditions $v(0) = 0$, $\frac{dv}{dt}(0) = \frac{2}{\sqrt{\pi}}$. With these initial conditions, Eq. (8.23) has a simple exact solution:

$$v(t) = \mathrm{Erf}(t), \tag{8.24}$$

where $\mathrm{Erf}(x)$ is defined as

$$\mathrm{Erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \mathrm{e}^{-y^2}\, dy. \tag{8.25}$$

Notice that the superior accuracy of the RK4 method very quickly becomes limited by round-off error. The smallest stepsize below which the error starts to get worse as the stepsize is decreased is quite modest - about 10^{-3} in this example.

8.7 Continuous Fourier Transform

The Fourier transform is used to represent a function as a sum of constituent harmonics. It is a linear invertible transformation between the time-domain representation of a function, which we shall denote by $h(t)$, and the frequency domain representation which we shall denote by $H(f)$. In one dimension, the Fourier transform pair consisting of the forward and inverse transforms, is often written as

$$\begin{aligned} H(f) &= \int_{-\infty}^{\infty} h(t)\, \mathrm{e}^{i\,f\,t}\, dt \\ h(t) &= \frac{1}{2\,\pi} \int_{-\infty}^{\infty} H(f)\, \mathrm{e}^{-i\,f\,t}\, df. \end{aligned}$$

Note that the Fourier transform is naturally defined in terms of complex functions. Both $h(t)$ and $H(f)$ are, in general, complex-valued. Normalisation can be a thorny issue however and there are many different conventions in different parts of the literature. In fact the transform pair

$$\begin{aligned} H(f) &= \sqrt{\frac{|b|}{(2\,\pi)^{1-a}}} \int_{-\infty}^{\infty} h(t)\, \mathrm{e}^{i\,b\,f\,t}\, dt \\ h(t) &= \sqrt{\frac{|b|}{(2\,\pi)^{1+a}}} \int_{-\infty}^{\infty} H(f)\, \mathrm{e}^{-i\,b\,f\,t}\, df. \end{aligned}$$

is equally good for any real values of the parameters a and b. In this module we shall adopt the convention $a = 0$ and $b = 2\,\pi$, which is common in the signal processing literature. With this choice the Fourier transform pair is:

$$\begin{aligned} H(f) &= \int_{-\infty}^{\infty} h(t)\, \mathrm{e}^{2\,\pi\,i\,f\,t}\, dt \\ h(t) &= \int_{-\infty}^{\infty} H(f)\, \mathrm{e}^{-2\,\pi\,i\,f\,t}\, df. \end{aligned} \tag{8.26}$$

The Fourier transform has several important symmetry properties which are often useful. The following properties are easily shown by direct calculation based on the definition:

- Parity is preserved: if $h(t)$ is even/odd then $H(f)$ is even/odd.
- Fourier transform of a real function: if $h(t)$ is real valued then $H(f)$, while still complex-valued, has the following symmetry:

  $$H(-f) = H(f)^*, \tag{8.27}$$

  where $H(f)^*$ denotes the complex conjugate of $H(f)$.

Sometimes it is possible to compute the integral involved in Eq. (8.26) analytically. Consider the important example of a Gaussian function:

$$h(t) = \frac{1}{\sqrt{2\,\pi\,\sigma^2}}\, \mathrm{e}^{-\frac{t^2}{2\,\sigma^2}}.$$

The Fourier transform can be calculated analytically using a standard trick which involves completing the square in the exponent of a Gaussian integral. From Eq. (8.26) we have

$$\begin{aligned} H(f) &= \frac{1}{\sqrt{2\,\pi\,\sigma^2}} \int_{-\infty}^{\infty} \mathrm{e}^{-\frac{t^2}{2\,\sigma^2}}\, \mathrm{e}^{2\,\pi\,i\,f\,t}\, dt \\ &= \frac{1}{\sqrt{2\,\pi\,\sigma^2}} \int_{-\infty}^{\infty} \mathrm{e}^{-\frac{1}{2\,\sigma^2}\left[t^2 - 4\pi i\sigma^2 t f\right]} dt \\ &= \frac{1}{\sqrt{2\,\pi\,\sigma^2}} \int_{-\infty}^{\infty} \mathrm{e}^{-\frac{1}{2\,\sigma^2}\left[t^2 - 2(2\pi i\sigma^2 f)t + (2\pi i\sigma^2 f)^2 - (2\pi i\sigma^2 f)^2\right]} dt \\ &= \frac{1}{\sqrt{2\,\pi\,\sigma^2}} \left[\int_{-\infty}^{\infty} \mathrm{e}^{-\frac{1}{2\,\sigma^2}(t - 2\pi i\sigma^2 f)^2} dt\right] \mathrm{e}^{\frac{1}{2\,\sigma^2}\,(2\pi i\sigma^2 f)^2} \\ &= \frac{1}{\sqrt{2\,\pi\,\sigma^2}} \left[\sqrt{2\,\pi\,\sigma^2}\right] \mathrm{e}^{-2\pi^2\sigma^2 f^2} \\ &= \mathrm{e}^{-2\pi^2\sigma^2 f^2}. \end{aligned}$$

Under the Fourier transform, the Gaussian function is mapped to another Gaussian function with a different width. If σ^2 is large/small then $h(t)$ is narrow/broad in the time domain. Notice how the width is inverted in the frequency domain. This is an example of a general principle: signals which are strongly localised in the time domain are strongly delocalised in the frequency domain and vice versa. This property of the Fourier transform is the underlying reason for the "Uncertainty Principle" in quantum mechanics where the "time" domain describes the position of a quantum particle and the "frequency" domain describes its velocity.

It is somewhat exceptional that the Fourier transform turns out to be a real quantity. In general, the Fourier transform, $H(f)$, of a real function, $h(t)$, is still complex. In fact, the Fourier transform of the Gaussian function is only real-valued because of the choice of the origin for the t-domain signal. If we would shift $h(t)$ in time, then the Fourier tranform would have come out complex. This can be seen from the following translation property of the Fourier transform. Suppose we define $g(t)$ to be a shifted copy of $h(t)$:

$$g(t) = h(t + \tau).$$

Writing both sides of this equation in terms of the Fourier transforms we have

$$\int_{-\infty}^{\infty} G(f)\, \mathrm{e}^{-2\,\pi\,i\,f\,t}\, df = \int_{-\infty}^{\infty} H(f)\, \mathrm{e}^{-2\,\pi\,i\,f\,(t+\tau)}\, df,$$

from which we conclude that

$$G(f) = \mathrm{e}^{-2\pi i\tau}\, h(f). \tag{8.28}$$

Translation in the time-domain therefore corresponds to multiplication (by a complex number) in the frequency domain.

8.7.1 Power Spectrum

Since Fourier transforms are generally complex valued, it is convenient to summarise the spectral content of a signal by plotting the squared absolute value of $H(f)$ as a function of f. This quantity,

$$P(f) = H(f)\, H^*(f), \tag{8.29}$$

is called the power spectrum of $h(t)$. The power spectrum is fundamental to most applications of Fourier methods.

8.7.2 Convolution Theorem

The convolution of two functions $f(t)$ and $g(t)$ is the function $(f * g)(t)$ defined by

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)\, g(t + \tau)\, d\tau. \tag{8.30}$$

Convolutions are used very extensively in time series analysis and image processing, for example as a way of smoothing a signal or image. The Fourier transform of a convolution takes a particularly simple form. Expressing f and g in terms of their Fourier transforms in Eq. (8.30) above, we can write

$$\begin{aligned}(f*g)(t) &= \int_{-\infty}^{\infty}\left[\int_{-\infty}^{\infty} F(f_1)\,\mathrm{e}^{-2\,\pi\,i\,f_1\,\tau}\,df_1\right]\left[\int_{-\infty}^{\infty} g(f_2)\,\mathrm{e}^{-2\,\pi\,i\,f_2\,(t+\tau)}\,df_2\right]\,d\tau \\ &= \int_{-\infty}^{\infty} df_1 \int_{-\infty}^{\infty} df_2\, F(f_1)\,G(f_2)\,\mathrm{e}^{-2\pi i f_1}\int_{-\infty}^{\infty} d\tau \mathrm{e}^{-2\pi i(f_1+f_2)} \\ &= \int_{-\infty}^{\infty} df_1 \int_{-\infty}^{\infty} df_2\, F(f_1)\,G(f_2)\,\mathrm{e}^{-2\pi i f_1}\,\delta(f_1+f_2) \\ &= \int_{-\infty}^{\infty} df_1 F(f_1)\,G^*(f_1)\,\mathrm{e}^{-2\pi i f_1}.\end{aligned}$$

Thus the Fourier transform of the convolution of f and g is simply the product of the individual Fourier transforms of f and g. This fact is sometimes called the Convolution Theorem. One of the most important uses of convolutions is in time-series analysis. The convolution of a signal, $h(t)$, (now thought of as a time-series) with itself is known as the auto-correlation function (ACF) of the signal. It is usually written

$$R(\tau) = \int_{-\infty}^{\infty} h(t+\tau)\,h(t)\,dt. \tag{8.31}$$

The ACF quantifies the memory of a time-series. Using the Convolution Theorem we can write

$$R(\tau) = \int_{-\infty}^{\infty} df H(f)\,H^*(f)\,\mathrm{e}^{-2\pi i f} = \int_{-\infty}^{\infty} df P(f)\,\mathrm{e}^{-2\pi i f}. \tag{8.32}$$

The ACF of a signal is the inverse Fourier transform of the power spectrum. This fundamental result is known as the Wiener-Kinchin Theorem. It reflects the fact that the frequency domain and time-domain representations of a signal are just different ways of looking at the same information.

The Convolution Theorem means that convolutions are very easy to calculate in the frequency domain. In order for this to be practically useful we must be able to do the integral in Eq. (8.26) required to return from the frequency domain to the time domain. As usual, the set of problems for which this integral can be done using analytic methods is a set of measure zero in the space of all potential problems. Therefore a computational approach to computing Fourier transforms is essential. Such an approach is provided by the Discrete Fourier Transform which we now discuss.

8.8 Discrete Fourier Transform

8.8.1 Time-domain discretisation and the Sampling Theorem

In applications, a time-domain signal, $h(t)$, is approximated by a set of discrete values, $\{h_n = h(t_n),\ n = \ldots -2, -$ of the signal measured at a list of consecutive points $\{t_n,\ n = \ldots -2, -1, 0, 1, 2, \ldots\}$. We shall assume that the sample points are uniformly spaced in time: $t_n = n\,\Delta$. The quantity Δ is called the sampling interval and its reciprocal, $1/\Delta$, is the sampling rate. The quantity

$$f_c = \frac{1}{2\Delta}. \tag{8.33}$$

is known as the Nyquist frequency. The importance of the Nyquist frequency will become apparent below.

Clearly, the higher the sampling rate, the more closely the sampled points, $\{h_n\}$, can represent the true underlying function. Of principal importance in signal processing is the question of how high the sampling rate should be in order to represent a given signal to a given degree of accuracy. This question is partially answered by a fundamental result known as the Sampling Theorem (stated here without derivation):

Theorem 8.8.1. *Let $h(t)$ be a function and $\{h_n\}$ be the samples of $h(t)$ obtained with sampling interval Δ. If $H(f) = 0$ for all $|f| \geq f_c$ then $h(t)$ is* completely *determined by its samples. An explicit reconstruction is provided by the Whittaker-Shannon interpolation formula:*

$$h(t) = \Delta \sum_{n=-\infty}^{\infty} h_n \frac{\sin\left[2\pi f_c \left(t - n\Delta\right)\right]}{\pi\left(t - n\Delta\right)}.$$

This theorem explains why the Nyquist frequency is important. There are two aspects to this. On the one hand, if the signal $h(t)$ is such that the power spectrum is identically zero for large enough absolute frequencies then the Sampling Theorem tells us that it is possible to reconstruct the signal perfectly from its samples. The Nyquist frequency tells us the minimal sampling rate which is required in order to do this. On the other hand, we know that most generic signals have a power spectrum whose tail extends to all frequencies. For generic signals, the amount of power outside the Nyquist interval, $[-f_c, f_c]$, limits the accuracy with which we can reconstruct a signal from its samples. The Nyquist frequency is then used in conjunction with an error tolerance criterion to decide how finely a function must be sampled in order to reconstruct the function from its samples to a given degree of accuracy.

8.8.2 Aliasing

The above discussion about using the Nyquist frequency to determine the sampling rate required to reconstruct a signal to a given degree of accuracy implicitly assumes that we know the power spectrum of $h(t)$ to begin with in order to determine how much power lies outside the Nyquist interval. In practice, we usually only have access to the samples of $h(t)$. From the samples alone, it is in principle impossible to determine how much power is outside the Nyquist interval. The reason for this is that the process of sampling moves frequencies which lie outside of the interval $[-f_c, f_c]$ into this interval via a process known as aliasing.

Aliasing occurs because two harmonics $\mathrm{e}^{2\pi i f_1 t}$ and $\mathrm{e}^{2\pi i f_2 t}$ give the same samples at the points $t_n = n\Delta$ if $f_1 - f_2$ is an integer multiple of Δ^{-1} (which is the width of the Nyquist interval). This can be seen by direct computation:

$$\begin{aligned}
&\mathrm{e}^{2\pi i f_1 n\Delta} = \mathrm{e}^{2\pi i f_1 n\Delta} && \forall n \\
\Rightarrow &\mathrm{e}^{2\pi i (f_1 - f_2) n\Delta} = 1 && \forall n \\
\Rightarrow &\mathrm{e}^{2\pi i (f_1 - f_2)\Delta} = 1 && \\
\Rightarrow &(f_1 - f_2)\,\Delta = m && m \in \mathbb{Z} \\
\Rightarrow &(f_1 - f_2) = \frac{m}{\Delta} && m \in \mathbb{Z}
\end{aligned}$$

Due to aliasing, it is impossible to tell the difference between the signals $h_1(t) = \mathrm{e}^{2\pi i f_1 t} + \mathrm{e}^{2\pi i f_2 t}$ and $h_2(t) = 2\,\mathrm{e}^{2\pi i f_1 t}$ from the samples alone. Aliasing means that under-sampled signals cannot be identified as such from looking at the sampled values. Care must therefore be taken in practice ensure that the sampling rate is sufficient to avoid aliasing errors.

8.8.3 Derivation of the Discrete Fourier Transform

Let us suppose we now have a finite number, N, of samples of the function $h(t)$: $\{h_n = h(t_n),\ n = 0, \ldots N-1\}$. For convenience we will assume that N is even. We now seek to use these values to estimate the integral in Eq. (8.26) at a set of discrete frequency points, f_n. What values should we choose for the f_n? We know from the Sampling Theorem that if the sampling rate is sufficiently high, the Nyquist interval contains the information necessary to reconstruct $h(t)$. If the sampling rate is too low, the power outside the Nyquist interval will be mapped back into the Nyquist interval by aliasing. In either case, we should choose our frequency points to discretise the Nyquist interval. Let us do this uniformly by choosing $f_n = \frac{n}{\Delta N}$ for $n = -\frac{N}{2}, \ldots + \frac{N}{2}$. The extremal frequencies are clearly $\pm f_c$. Those paying close attention will notice that there are $N+1$ frequency points to be estimated from N samples. It will turn out that not all $N+1$ frequencies are independent. We will return to this point in a moment.

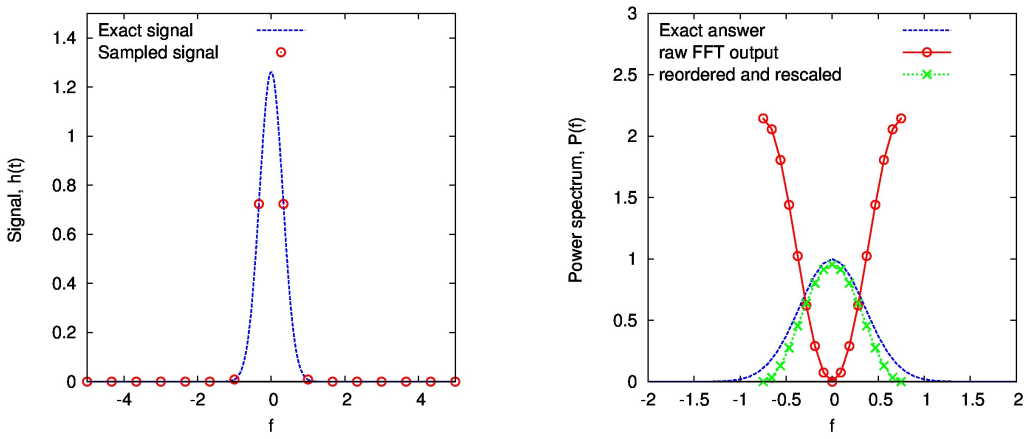


Figure 8.5: Discrete Fourier Transform of a Gaussian sampled with 16 points in the domain [-5:5] using the fftw3 implementation of the Fast Fourier Transform.

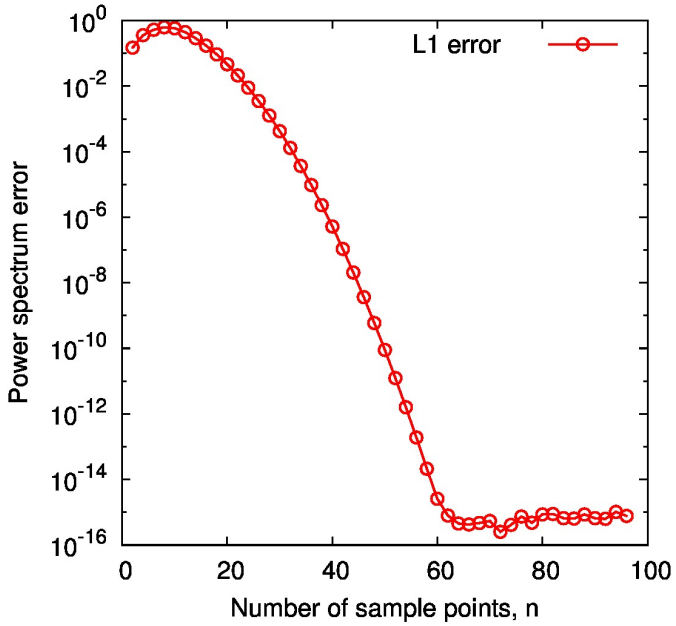


Figure 8.6: L1 error in the power spectrum of a Gaussian approximated using the Discrete Fourier Transform plotted as a function of the number of sample points in the domain [-5:5]. The fftw3 implementation of the Fast Fourier Transform was used.

Let us now estimate the integral $H(f)$ in Eq. (8.26) at the discrete frequency points f_n by approximating the integral by a Riemann sum:

$$\begin{aligned} H(f_n) &\approx \sum_{k=0}^{N-1} h_k \, \mathrm{e}^{2\pi i f_n t_k} \Delta \\ &= \Delta \sum_{k=0}^{N-1} h_k \, \mathrm{e}^{2\pi i \left(\frac{n}{N\Delta}\right)(k\Delta)} \\ &= \Delta \sum_{k=0}^{N-1} h_k \, \mathrm{e}^{\frac{2\pi i n k}{N}} \\ &\equiv \Delta \, H_n, \end{aligned}$$

where the array of numbers, $\{H_n\}$, defined as

$$H_n = \sum_{k=0}^{N-1} h_k \, \mathrm{e}^{\frac{2\pi i n k}{N}} \tag{8.34}$$

is called the Discrete Fourier Transform (DFT) of the array $\{h_n\}$. The inverse of the transform is obtained by simply changing the sign of the exponent. Note that the DFT is simply a map from one array of complex numbers to another. All information about the time and frequency scales (including the sampling rate) have been lost. Thought of as a function of the index, n, the DFT is periodic with period N. That is

$$H_{n+N} = H_n.$$

We can see this by direct computation:

$$\begin{aligned} H_{n+N} &= \sum_{k=0}^{N-1} h_k \, \mathrm{e}^{\frac{2\pi i (n+N) k}{N}} \\ &= \sum_{k=0}^{N-1} h_k \, \mathrm{e}^{\frac{2\pi i n k}{N}} \, \mathrm{e}^{2\pi i k} \\ &= \sum_{k=0}^{N-1} h_k \, \mathrm{e}^{\frac{2\pi i n k}{N}} \\ &= H_n. \end{aligned}$$

We can therefore shift the index, n, used to define the frequencies f_n, by $\frac{N}{2}$ in order to obtain an index which runs over the range $n = 0 \ldots N$ rather than $n = -\frac{N}{2}, \ldots + \frac{N}{2}$. This is more convenient for a computer where arrays are usually indexed by positive numbers. Most implementations of the DFT adopt this convention. One consequence of this convention is that if we want to plot the DFT as a function of the frequencies (for example to compare against an analytic expression for the corresponding continuous Fourier transform) we need to obtain the negative frequencies from $H_{-n} = HN - n$. Another consequence of this periodicity is that $H_{-\frac{N}{2}} = H_{\frac{N}{2}}$ so $H_{\frac{N}{2}}$ corresponds to both frequencies $f = \pm f_c$ at either end of the Nyquist interval. Thus, as mentioned above, only N of the $N+1$ values of n chosen to discretise the Nyquist interval are independent. THe DFT, Eq. (8.34), is therefore a map between arrays of length N which can be written as a matrix multiplication:

$$H_n = \sum_{k=0}^{N-1} W_{nk} \, h_k, \tag{8.35}$$

where W_{pq} are the elements of the $N \times N$ matrix

$$W_{pq} = \mathrm{e}^{\frac{2\pi i p q}{N}}.$$

The application of the DFT to a sampled version of the Gaussian function is shown in Fig. 8.5. The frequency domain plot on the right of Fig. 8.5 shows the power spectrum obtained from the output of the DFT plotted simply as a function of the array index (in red). Notice how the frequencies appear in the "wrong" order as a result of the shift described above. Notice also how the amplitude of the power spectrum obtained from the DFT is significantly larger than the analytic expression obtained for the continuous Fourier transform. This is due to the fact that the pre-factor of Δ in the approximation of $H(f)$ derived above is not included in the definition of the DFT. When both of these facts are taken into account by rescaling and re-ordering the output of the DFT, the green points are obtained which are a much closer approximation to the true power spectrum. The difference between the two is due to the low sampling rate. As the sampling rate is increased the approximation obtained from the DFT becomes better. This is illustrated in Fig. 8.6 which plots the error as a function of the number of sample points. Why do you think the error saturates at some point?

8.9 The Fast Fourier Transform algorithm

From Eq. (8.35), one might naively expect that the computation of the DFT is an $O(N^2)$ operation. In fact, due to structural regularities in the matrix W_{pq} it is possible to perform the computation in $O(N \log_2 N)$ operations using a divide-and-conquer algorithms which is known as the Fast Fourier Transform (FFT). In all that follows we will assume that the input array has length $N = 2^L$ elements. The FFT algorithm is based on the Danielson-Lanczos lemma which states that a DFT of length N can be written as a sum of 2 DFTs of length $N/2$. This can be seen by direct computation where the trick is to build the sub-transforms of lenght $N/$ from the elements of the original transform whose indices are odd/even respectively:

$$\begin{aligned} H_k &= \sum_{j=0}^{N-1} h_j \, \mathrm{e}^{\frac{2\pi i n j}{N}} \\ &= \left[\sum_{j=0}^{\frac{N}{2}-1} h_{2j} \, \mathrm{e}^{\frac{2\pi i n (2j)}{N}} \right] + \left[\sum_{j=0}^{\frac{N}{2}-1} h_{2j+1} \, \mathrm{e}^{\frac{2\pi i n (2j+1)}{N}} \right] \\ &= \left[\sum_{j=0}^{\frac{N}{2}-1} h_j^{(e)} \, \mathrm{e}^{\frac{2\pi i n j}{N/2}} \right] + \mathrm{e}^{\frac{2\pi i k}{N}} \left[\sum_{j=0}^{\frac{N}{2}-1} h_j^{(o)} \, \mathrm{e}^{\frac{2\pi i n j}{N/2}} \right]. \end{aligned}$$

where $\left\{ h_j^{(o/e)}, j = 0, \ldots \frac{N}{2} - 1 \right\}$ are the arrays of length $\frac{N}{2}$ constructed from the odd/even elements of the original input array respectively. The last line is simply a linear combination of the DFTs of these two arrays:

$$H_k = H_k^{(e)} + \alpha_k \, H_k^{(o)}, \tag{8.36}$$

where $\alpha_k = \mathrm{e}^{\frac{2\pi i k}{N}}$. Note that since the index k on the LHS ranges over $k = 0, \ldots, N-1$, whereas the arrays on the RHS are only of length $N/2$, the periodicity of the DFT must be invoked to make sense of this formula. The FFT algorithm works by recognising that this can be applied recursively. After $L = \log_2 N$ subdivisions, we are left with arrays of length 1, each of which contains one of the elements of the original input array. Thus at the lowest level of the recursion, we just have a re-ordered copy of the original array. This is illustrated schematically in Fig. 8.7. The DFT of an array of length one is trivially just the array itself. The DFT of the original array can therefore be reassembled by going back up the tree linearly combining pairs of arrays of length n at each level to form arrays of length $2n$ at the next level up according to the rule given in Eq. (8.36 until we arrive back at the top level. At each level, there are clearly N operations required and there are $L = \log_2 N$ levels. Therefore the total complexity of this process is $O(N \log_2 N)$.

The key insight of the FFT algorithm is to construct the re-ordered array first and then go back up the tree. Notice how each position in the array at the bottom level can be indexed by a string of e's and o's depending on the sequence of right and left branches which are followed when descending the tree to reach that position. This sequence can be reversed to find out which element of the input array should be placed at which position at the bottom of the recursion tree. This is done using the

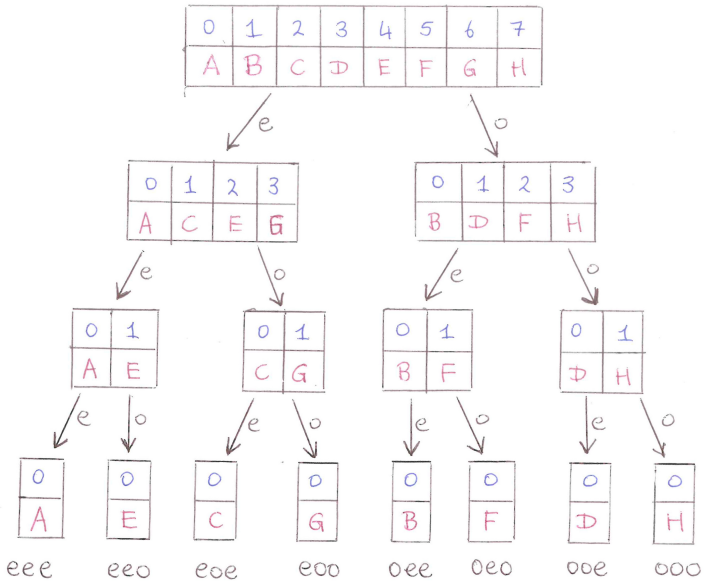


Figure 8.7: Graphical illustration of the recursive structure of the FFT algorithm for an array of length 8. At each level, each array is divided into two sub-arrays containing the elements with odd and even indices respectively. At the lowest level, the position of an element is indexed by a unique string of e's and o's. This string is determined by the sequence of right and left branches which are followed when descending the tree to reach that position. The bit reversal trick then identifies which element of the original array ends up in each position.

bit-reversal rule. Starting with the string corresponding to a particular position at the bottom of the recursion tree we proceed as follows:

1. reverse the order of the string of e's and o's.
2. replace each e with 0 and each o with 1.
3. interpret the resulting string of zeros and ones as the binary representation of an integer.

The integer obtained in this way is the index of the element in the original input array which should be placed at this position at the bottom of the recursion tree. It is worth convincing oneself that this really works by checking some explicit examples in Fig. 8.7. This re-ordering can be done in $O(N)$ operations.

Bibliography

[1] Linear multistep method, 2014. http://en.wikipedia.org/wiki/Linear_multistep_method.

[2] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Numerical recipes: The art of scientific computing (3rd ed.). Cambridge University Press, New York, NY, USA, 2007.

[3] C. Moler. Matlab news & notes may 2003 - stiff differential equations, 2003. http://www.mathworks.com/company/newsletters/news_notes/clevescorner/may03_cleve.html.

[4] L. F. Shampine and S. Thompson. Stiff systems. Scholarpedia, 2(3):2855, 2007.