9 General scientific programming

9.1 Floating point arithmetic

9.1.1 The representation of real numbers

The real numbers, such as 1.2, -0.36, π, 4 and 13256.625 may be thought of as points on a continuous, infinite *number line*.[1] Some real numbers (including the integers themselves) can be expressed as a ratio of two integers, for example, $\frac{5}{8}$ and $\frac{1}{3}$. Such numbers are called *rational*. Others, such as π, e and $\sqrt{2}$ cannot and are called *irrational*.

There can therefore be several ways of writing a real number, depending on which category it falls into, and not all of these ways can express the number precisely (using a finite amount of ink!). For example, the rational real number $\frac{5}{8}$ may be written exactly as a *decimal expansion* as 0.625:

$$\frac{5}{8} = \frac{6}{10} + \frac{2}{100} + \frac{5}{1000},$$

but the number $\frac{1}{3}$ cannot be written in a finite number of terms of a decimal expansion:

$$\frac{1}{3} = \frac{3}{10} + \frac{3}{100} + \frac{3}{1000} + \cdots = 0.333\cdots$$

In writing $\frac{1}{3}$ as a decimal expansion we must truncate the infinite sequence of 3s somewhere.

The irrational numbers can be *described* exactly (given some presumed geometrical or other knowledge), for example, π is the ratio of a circle's circumference to its diameter, $\sqrt{2}$ is the length of the hypotenuse of a right-angled triangle whose other sides have length 1. To *represent* or store such a number numerically, however, some level of approximation is necessary. For example, $\frac{355}{113}$ is a famous rational approximation to π. A (better) decimal approximation is 3.14159265358979. But, as a decimal expansion,[2] an infinite number of digits are needed to express the value of π precisely, just as an infinite number of 3s are needed in the decimal expansion of $\frac{1}{3}$.

Computers store numbers in binary, and the same considerations that apply to the limits of the decimal representation of a real number apply to its binary representation.

[1] Obviously, an integer such as 4 is just a special sort of real number.

[2] Note that a decimal expansion is simply a rational number with a power of 10 in the denominator, $3.14159265358979 = \frac{314159265358979}{100000000000000}$.

For example, $\frac{5}{8}$ has an exact binary representation in a finite number of bits:

$$\frac{5}{8} = \frac{1}{2} + \frac{0}{4} + \frac{1}{8} = 0.101_2$$

but

$$\frac{1}{10} = 0.0001100110011001100110011\cdots_2\,,$$

an infinitely repeating sequence. Only a finite number of these digits can be stored, and the truncated series of bits converted back to decimal is

$$\frac{1}{10} \approx 0.10000000000000009$$

using the so-called *double-precision* standard common to most computer languages on most operating systems. This is the *nearest representable number* to $\frac{1}{10}$.

The format of the double-precision floating point representation of numbers is dictated by the IEEE-754 standard. There are three parts to the representation, stored in a total of 64 bits (8 bytes): the single *sign bit*, an 11-bit *exponent* and a 52-bit *significand* (also called the *mantissa*). This is best demonstrated by an example in decimal: the number 13256.625 can be written in scientific notation as:

$$13256.625 = +1.3256625 \times 10^4$$

and stored with the sign bit corresponding to `+`, a significand equal to `13256625` (where the decimal point is implicitly to be placed after the first digit) and the exponent `4`. This notation is called "floating point" because the decimal point[3] is moved by the number of places indicated by the exponent.

The floating point representation of numbers in binary works in the same way, except that each digit can only be 0 or 1, of course. This allows for a neat trick: when the number's binary point (equivalent to the decimal point in base-10) is shifted so that its significand has no leading zeros, then it will start with 1. Because all significands *normalized* in this way will start with 1, there is no need to store it, and effectively 53 bits of precision can be stored in a 52-bit significand.[4] The omitted bit is sometimes called the *hidden bit*.

In our example, 13256.625 can in fact be represented exactly in binary as

$$13256.625_{10} \equiv 11001111001000.101_2$$

The normalized form of the significand is therefore `1100111100100010 1` and the exponent is 13, since:

$$11001111001000.101_2 = 1.1001111001000101 \times 2^{13}.$$

Now, as discussed, the first digit of the normalized signifcand will always be 1, so it is omitted and the significand is stored as

```
1001111001000101000000000000000000000000000000000000
```

[3] More generally known as the *radix* point in bases other than base-10.

[4] Note that this trick works only in the binary sysetm.

In order to allow for negative exponents (numbers with magnitudes less than 1), the exponent is stored with a *bias*: 1023 is added to the actual exponent. That is, actual exponents in the range -1022 to $+1023$ are stored as numbers in the range 1 to 2046. In this case, the 11-bit exponent field is $13 + 1023 = 1036$:

```
10000001100
```

Finally, the sign bit is `0`, indicating a positive number. The full, 64-bit floating point representation of 13256.625 (with spaces for clarity) is

```
0 10000001100 1001111001000101000000000000000000000000000000000000000000
```

and is exact. However, 0.1 is

```
0 01111111011 1001100110011001100110011001100110011001100110011001100110011010
```

and is not exact (note the truncation and rounding of the infinitely repeating sequence 0011...) – in decimal, this number is

```
0.1000000000000000055511151231257
```

In general, the 53 bits (including the *hidden bit*) of the significand give about 15 decimal digits of precision ($\log_{10}(2^{53}) = 15.95$). Any calculation resulting in more significant digits is subject to *rounding error*. The upper bound of the relative error due to rounding is called the *machine epsilon*, ϵ. In Python,

```
In [x]: import sys
In [x]: eps = sys.float_info.epsilon
In [x]: eps
Out[x]: 2.220446049250313e-16
```

It can be shown that the maximum spacing between two normalized floating point numbers is 2ϵ. That is, `x + 2*eps == x` is guaranteed always to be `False`.

9.1.2 Comparing floating point numbers

Because of the finite precision of the floating point representation of (most) real numbers it is extremely risky to compare two `floats` for equality. For example, consider squaring 0.1:

```
In [x]: (0.1)**2
Out[x]: 0.010000000000000002
```

As we have come to expect, this is not exactly 0.01, but it is also *not even the nearest representable number to 0.01*, since the number squared was, in fact, 0.10000000000000009. The unfortunate consequence of this is

```
In [x]: (0.1)**2 == 0.01
Out[x]: False
```

NumPy provides the methods `isclose` and `allclose` (see Section 6.1.12) for comparing two floating point numbers or arrays to within a specified or default tolerance:

```
In [x]: np.isclose(0.1**2, 0.01)
Out[x]: True
```

Note also that floating point addition is not necessarily *associative*:

```
In [x]: a, b, c = 1e14, 25.44, 0.74
In [x]: (a + b) + c
Out[x]: 100000000000026.17

In [x]: a + (b + c)
Out[x]: 100000000000026.19
```

Nor, in general, is floating point multiplication *distributive* over addition:

```
In [x]: a, b, c = 100, 0.1, 0.2

In [x]: a*b + a*c
Out[x]: 30.0

In [x]: a * (b + c)
Out[x]: 30.000000000000004
```

9.1.3 Loss of significance

Most floating point operations (such as addition and subtraction) result in a loss of significance. That is, the number of significant digits in the result can be smaller than in the original numbers (operands) used in the calculation. To illustrate this, consider a hypothetical floating point representation working in decimal with a 6-digit significand and perform the following calculation, written in its exact form:

$$1.2345432 - 1.23451 = 0.0000332$$

Our hypothetical system cannot store the first operand to its full precision but can only get as close as 1.23454. The floating point subtraction then yields

$$1.23454 - 1.23451 = 0.00003.$$

The original numbers were accurate in the most significant six digits, but the result is only accurate in its first significant digit. Note that it isn't the case that the exact result cannot be *represented* in all its digits by our floating point architecture: $0.0000332 \equiv 3.32 \times 10^{-5}$ only has three significant digits, well within the six available to us. The drastic loss of significance occurred because there was only a very small difference between the two numbers. This effect is sometimes called *catastrophic cancellation* and should always be a consideration when subtracting two numbers with similar values.

A similar loss of significance can occur when a small number is subtracted from (or added to) a much larger one:

$$\begin{aligned} &12345.6 + 0.123456 = 12345.72345 && \text{(exactly)}, \\ &12345.6 + 0.123456 = 12345.7 && \text{(6-digit decimal significand)}. \end{aligned}$$

Even though the 15 or so significant digits of a double-precision floating point number may seem like sufficient accuracy for a single calculation, be aware that repeatedly carrying out such calculations can increase this rounding error dramatically if the numbers involved cannot be represented exactly. For example, consider the following:

```
In [x]: for i in range(10000000):
   ....:     a += 0.1
   ....:

In [x]: a
Out[x]: 999999.9998389754
```

The difference between this approximate value and the exact answer, 1000000, is over 1.61×10^{-4}.

Python's `math` module has a function, `fsum`, which uses a technique called the *Shewchuk algorithm* to compensate for rounding errors and loss of significance. Compare these two implementations of the previous sum using a generator expression:

```
In [x]: sum((0.1 for i in range(10000000)))
Out[x]: 999999.9998389754
In [x]: math.fsum((0.1 for i in range(10000000)))
Out[x]: 1000000.0
```

9.1.4 Underflow and overflow

Another consequence of the way that floating point numbers are handled is that there is a minimum and maximum magnitude of number that can be stored. For example, Bayesian calculations frequently require small probabilities to be multiplied together, with each probability a number between 0 and 1. For a large number of such probabilities this product can reach a value that is too small to represent resulting in *underflow* to zero:

```
In [x]: P = 1
In [x]: for i in range(101):
  ....:     print(P)
  ....:     P *= 5.e-4

1
0.0005
2.5e-07
1.25e-10
6.250000000000001e-14
...
1.0097419586828971e-307
5.0487097934146e-311        # denormalization starts
2.5243548965e-314
1.2621776e-317
6.31e-321
5e-324
0.0                         # underflow
0.0
```

Below this value, Python begins to sacrifice some of the precision and maintains a modified representation of the number (a denormal, or subnormal number), a process called *gradual underflow*. Eventually, however, the number underflows its representation totally and becomes indistinguishable from zero. The minimum number that can be represented at full IEEE-754 double precision is

```
In [x]: import sys
In [x]: sys.float_info.min
Out[x]: 2.2250738585072014e-308
```

There are several possible tactics for dealing with underflow (beyond using higher precision numbers such as `np.float128`). In the earlier example, it is common to take the sum of the logarithms of the probabilities, which has a much more modest magnitude, instead of taking the product directly. Alternatively, one could start the earlier code with `P = 1.e100` and manipulate the resulting numbers on the understanding that they are larger than they should be by this constant factor.

Floating point *overflow* is the problem at the other end of the number scale: the largest double-precision number that can be represented is

```
In [x]: sys.float_info.max
Out[x]: 1.7976931348623157e+308
```

In NumPy, numbers that overflow are set to the special values `inf` or `-inf` depending on sign:

```
In [x]: f = 1
In [x]: for x in range(1,40,4):
   ...:     print('exp({}) = {}'.format(x**2, np.exp(x**2)))
   ...:
exp(1) = 2.718281828459045
exp(25) = 72004899337.38588
exp(81) = 1.5060973145850306e+35
exp(169) = 2.487524928317743e+73
exp(289) = 3.2441824460394912e+125
exp(441) = 3.340923407659982e+191
exp(625) = 2.7167594696637367e+271
exp(841) = inf
exp(1089) = inf
exp(1369) = inf
```

This leads to some curious relations between numbers that are too big to represent:

```
In [x]: a, b = 1.e500, 1.e1000
In [x]: a == b
Out[x]: True
In [x]: a, b
Out[x]: (inf, inf)
```

There is another special value, `nan` ("not-a-number", NaN), which is returned by some operations involving overflowed numbers:

```
In [x]: a / b
Out[x]: nan
```

(NumPy also implements its own values, `numpy.nan` and `numpy.inf`, see Section 6.1.4.) Never check if an object is `nan` with the `==` operator: `nan` is not even equal to itself(!)[5]:

[5] Note that this means that the `==` operator is not an *equivalence relation* for floating point numbers as it is not reflexive.

```
In [x]: c = a / b
In [x]: c == c
Out[x]: False
```

Python `int` objects are not subject to overflow, as Python will automatically allocate memory to hold them to full precision (within the limitations of available machine memory). However, NumPy integer arrays, which map to the underlying C data structures are stored in a fixed number of bytes (see Table 6.2) and may overflow. For example,

```
In [x]: a = np.zeros(3, dtype=np.int16)
In [x]: a[:] = -30000, 30000, 40000
In [x]: a
Out[x]: array([-30000,  30000, -25536], dtype=int16)

In [x]: b = np.zeros(3, dtype=np.uint16)
In [x]: b[:] = -30000, 40000, 70000
In [x]: b
Out[x]: array([35536, 40000,  4464], dtype=uint16)
```

Signed 16-bit integers have the range -32768 to 32767 (-2^{15} to $(2^{15}-1)$). Due to the way they are stored, an attempted assignment to the number 40000 has resulted instead in the assignment of $40000-2^{16}=-25536$ to `a[2]` above. Similarly, *unsigned* 16-bit integers are limited to values in the range 0 to 65535 (0 to $(2^{16}-1)$). Negative numbers cannot be represented at all and `b[0] = -30000` gets converted to $-30000 \mod 2^{16} = 35536$; `b[2] = 70000` overflows and ends up as $70000 \mod 2^{16} = 4464$.

9.1.5 Further Reading

- From the Python documentation: *Floating Point Arithmetic: Issues and Limitations*, available at http://docs.python.org/tutorial/floatingpoint.html.
- The article "What Every Computer Scientist Should Know About Floating-Point Arithmetic" by David Goldberg (*Computing Surveys*, March 1991) has become something of a classic and for a rigorous approach to the topic of floating point arithmetic is highly recommended. It is available at https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html.
- S. Oliveira and D. Stewart, (2006).*Writing Scientific Software: A Guide to Good Style*, Cambridge University Press.
- N. J. Higham, (2002). *Accuracy and Stability of Numerical Algorithms*, 2nd ed., Society for Industrial and Applied Mathematics.

9.1.6 Exercises

Questions

Q9.1.1 The *decimal representation* of some real numbers is not unique. For example, prove mathematically that $0.\dot{9} \equiv 0.9999\cdots \equiv 1$.

Q9.1.2 $\sqrt{\tan(\pi)} = 0$ is mathematically well-defined, so why does the folllowing calculation fail with a math domain error?

```
In [x]: math.sqrt(math.tan(math.pi))
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-135-7bfdceeef434> in <module>()
----> 1 math.sqrt(math.tan(math.pi))
```

Q9.1.3 Fermat's Last Theorem states that no three positive integers x, y and z can satisfy the equation $x^n + y^n - z^n = 0$ for any integer $n > 2$. Explain this apparent counter-example to the theorem:

```
In [x]: 844487.**5 + 1288439.**5 - 1318202.**5
Out[x]: 0.0
```

Q9.1.4 The functions $f(x) = (1 - \cos^2 x)/x^2$ and $g(x) = \sin^2 x/x^2$ are mathematically indistinguishable, but plotted using Python in the region $-0.001 \leq x \leq 0.001$ show a significant difference. Explain the origin of this difference.

Q9.1.5 How can you establish whether a floating point number is `nan` or not without using `math.isnan` or `numpy.isnan`?

Q9.1.6 Predict and explain the outcome of the following:

a. `1e1001 > 1e1000`
b. `1e350/1.e100 == 1e250`
c. `1e250 * 1.e-250 == 1e150 * 1.e-150`
d. `1e350 * 1.e-350 == 1e450 * 1.e-450`
e. `1 / 1e250 == 1e-250`
f. `1 / 1e350 == 1e-350`
g. `1e450/1e350 != 1e450 * 1e-350`
h. `1e250/1e375 == 1e-125`
i. `1e35 / (1e1000 - 1e1000) == 1 / (1e1000 - 1e1000)`
j. `1e1001 > 1e1000 or 1e1001 < 1e1000`
k. `1e1001 > 1e1000 or 1e1001 <= 1e1000`

Problems

P9.1.1 *Heron's formula* for the area of a triangle (as used in Example E2.3)

$$A = \sqrt{s(s-a)(s-b)(s-c)} \text{ where } s = \tfrac{1}{2}(a+b+c)$$

is inaccurate if one side is very much smaller than the other two ("needle-shaped" triangles). Why? Demonstrate that the following reformulation gives a more accurate result in this case by considering the triangle with sides $(10^{-13}, 1, 1)$, which has the area 5×10^{-14}.[6]

$$A = \frac{1}{4}\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))},$$

where the sides have been relabeled so that $a \geq b \geq c$.

[6] This formula is due to William Kahan, one of the designers of the IEEE-754 floating point standard.

What happens if you rewrite the factors in this equation to remove their inner parentheses? Why?

P9.1.2 Write a function to determine the machine epsilon of a numerical data type (`float`, `np.float128`, `int`, etc.).

9.2 Stability and conditioning

9.2.1 The stability of an algorithm

The stability of an algorithm may be thought of in relation to how it handles approximation errors that occur in its operation or its input data. These errors typically arise from experimental uncertainties (imperfect measurements providing the input data) or from the sort of floating point approximations involved in the calculations of the algorithm discussed in the previous section. Another common source of error is in the approximations made in "discretizing" a problem: the need to represent the values of a continuous function, $y = f(x)$ say, on a discrete "grid" of points: $y_i = f(x_i)$. An algorithm is said to be numerically stable if it does not magnify these errors and unstable if it causes them to grow.

Example E9.1 Consider the differential equation,

$$\frac{\mathrm{d}y}{\mathrm{d}x} = -\alpha y$$

for $\alpha > 0$ subject to the boundary condition $y(0) = 1$. This simple problem can be solved analytically:

$$y = e^{-\alpha x},$$

but suppose we want to solve it numerically. The simplest approach is the *forward* (or *explicit*) *Euler* method: choose a step size, h, defining a grid of x values, $x_i = x_{i-1} + h$, and approximate the corresponding y values through:

$$y_i = y_{i-1} + h \left| \frac{\mathrm{d}y}{\mathrm{d}x} \right|_{x_{i-1}} = y_{i-1} - h\alpha y_{i-1} = y_{i-1}(1 - \alpha h).$$

The question arises: what value should be chosen for h? A small h minimizes the error introduced by the approximation above which basically joins y values by straight-line segments,[7] but if h is too small there will be cancellation errors due to the finite precision used in representing the numbers involved.[8]

---

[7] That is, the Taylor series about y_{i-1} has been truncated at the linear term in h.

[8] In the extreme case that h is chosen to be smaller than the *machine epsilon*, typically about 2×10^{-16}, then we have $x_i = x_{i-1}$ and there is no grid of points at all.

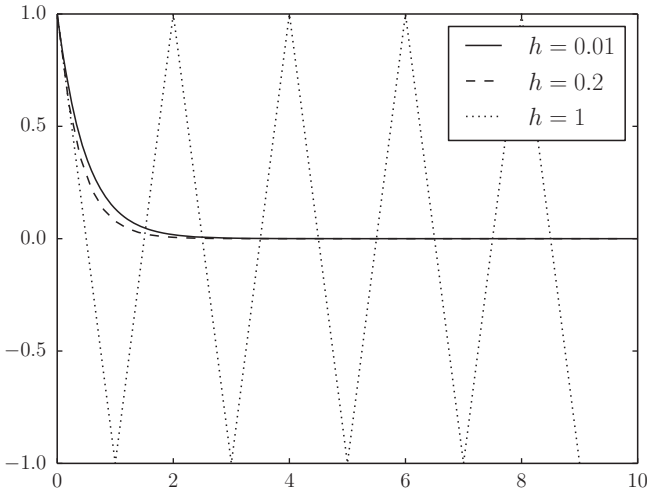

Figure 9.1 Instability of the forward-Euler solution to $\mathrm{d}y/\mathrm{d}x = -\alpha y$ for large step size, h.

The following code implements the forward Euler algorithm to solve the earlier differential equation. The largest value of h (here, $h = \alpha/2 = 1$) clearly makes the algorithm unstable (see Figure 9.1).

Listing 9.1 Comparison of different step sizes, h, in the numerical solution of $y' = -\alpha y$ by the forward Euler algorithm

```
import numpy as np
import pylab

alpha, y0, xmax = 2, 1, 10

def euler_solve(h, n):
    """ Solve dy/dx = -alpha.y by forward Euler method for step size h."""
    y = np.zeros(n)
    y[0] = y0
    for i in range(1, n):
        y[i] = (1 - alpha * h) * y[i-1]
    return y

def plot_solution(h):
    x = np.arange(0, xmax, h)
    y = euler_solve(h, len(x))
    pylab.plot(x, y, label='$h={}$'.format(h))

for h in (0.01, 0.2, 1):
    plot_solution(h)

pylab.legend()
pylab.show()
```

Example E9.2 The integral

$$I_n = \int_0^1 x^n e^x \, \mathrm{d}x \quad n = 0, 1, 2, \cdots$$

suggests a recursion relation obtained by integration by parts:

$$I_n = \left[x^n e^x\right]_0^1 - n \int_0^1 x^{n-1} e^x \, \mathrm{d}x = e - nI_{n-1}$$

terminating with $I_0 = e - 1$. However, this algorithm, applied "forward" for increasing n is numerically unstable since small errors (such as floating point rounding errors) are magnified at each step: if the error in I_n is ϵ_n such that the estimated value of $I'_n = I_n + \epsilon_n$ then

$$\epsilon_n = I'_n - I_n = (e - nI'_{n-1}) - (e - nI_{n-1}) = n(I_{n-1} - I'_{n-1}) = -n\epsilon_{n-1},$$

and hence $|\epsilon_n| = n!\epsilon_0$. Even if the error in ϵ_0 is small, that in ϵ_n is larger by a factor $n!$, which can be huge.

The numerically stable solution, in this case, is to apply the recursion backward for decreasing n:

$$I_{n-1} = \frac{1}{n}(e - I_n) \quad \Rightarrow \epsilon_{n-1} = -\frac{\epsilon_n}{n}.$$

That is, errors in I_n are *reduced* on each step of the recursion. One can even start the algorithm at $I'_N = 0$ and providing enough steps are taken between N and the desired n it will converge on the correct I_n.

Listing 9.2 Comparison of algorithm stability in the calculation of $I(n) = \int_0^1 x^n e^x \, \mathrm{d}x$

```
# eg9-integral-stability.py
import numpy as np
import pylab

def Iforward(n):
    if n == 0:
        return np.e - 1
    return np.e - n * Iforward(n-1)

def Ibackward(n):
    if n >= 99:
        return 0
    return (np.e - Ibackward(n+1)) / (n+1)

N = 35
Iforward = [np.e - 1]
for n in range(1, N+1):
    Iforward.append(np.e - n * Iforward[n-1])

Ibackward = [0] * (N+1)
for n in range(N-1,-1,-1):
    Ibackward[n] = (np.e - Ibackward[n+1]) / (n+1)

n = range(N+1)
pylab.plot(n, Iforward, label='Forward algorithm')
```

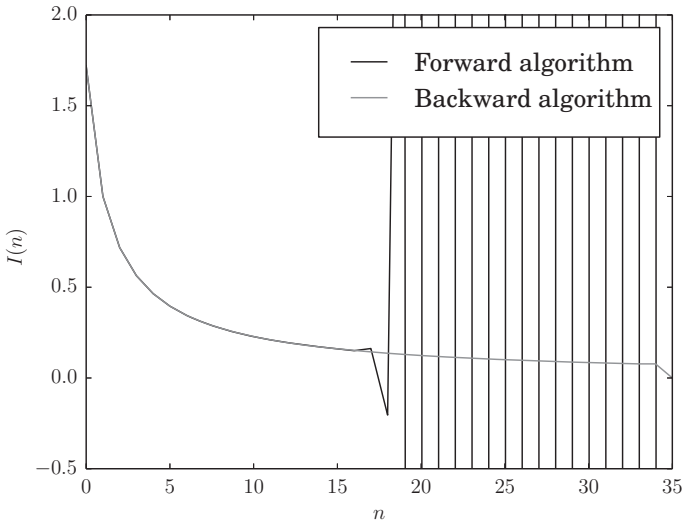

Figure 9.2 Instability of the forward recursion relation for $I_n = \int_0^1 x^n e^x \, \mathrm{d}x$.

```
pylab.plot(n, Ibackward, label='Backward algorithm')
pylab.ylim(-0.5, 2)
pylab.xlabel('$n$')
pylab.ylabel('$I(n)$')
pylab.legend()
pylab.show()
```

Figure 9.2 shows the forward algorithm becoming extremely unstable for $n > 16$ and fluctuating between very large positive and negative values; conversely, the backward algorithm is well behaved.

9.2.2 Well-conditioned and ill-conditioned problems

In numerical analysis, a further distinction is made between problems which are well- or ill-conditioned. A *well-conditioned problem* is one for which small relative errors in the input data lead to small relative errors in the solution; an *ill-conditioned problem* is one for which small input errors lead to large errors in the solution. Conditioning is a property of the problem, not the algorithm and is distinct from the issue of stability: it is perfectly possible to use an unstable algorithm on a well-conditioned problem and end up with erroneous results.

Example E9.3 Consider the two lines given by the equations:

$$y = x$$
$$y = mx + c$$

These lines intersect at $(x_\star, y_\star) = (c/(1-m), c/(1-m))$. Finding the intersection point is an ill-conditioned problem when $m \approx 1$ (lines nearly parallel).

For example, the lines $y = x$ and $y = (1.01)x+2$ intersect at $(x_\star, y_\star) = (-200, -200)$. If we perturb m slightly by $\delta m = 0.001$, to $m' = m+\delta m = 1.011$, the intersection point becomes $(x'_\star, y'_\star) = (-181.8182, -181.8182)$. That is, a relative error of $\delta m/m \approx 0.001$ in m has created a relative error of $|(x'_\star - x_\star)/x_\star| \approx 0.091$, almost 100 times larger.

Conversely, if the lines have very different gradients, the problem is well-conditioned. Take, for example, $m = -1$ (perpendicular lines): the intersection $(1, 1)$ becomes $(1.0005, 1.0005)$ under the same perturbation to $m' = m + \delta m = -0.999$, leading to a relative error of 0.0005, which is actually *smaller* than the relative error in m.

Example E9.4 The conditioning of polynomial root-finding is notoriously bad. One famous example is *Wilkinson's polynomial*:

$$\begin{aligned} P(x) &= \prod_{i=1}^{20}(x-i) = (x-1)(x-2)\cdots(x-20) \\ &= x^{20} - 210x^{19} + 20615x^{18} + \cdots + 2432902008176640000 \end{aligned}$$

By inspection, the roots are simply $1, 2, \cdots, 20$. However, Wilkinson showed that decreasing the coefficient of x^{19} from -210 to $-210 - 2^{-23} \approx -210.000000119209$ had a drastic effect on many of the roots, some of which become complex. For example, the root at $x = 20$ moves to $x = 20.8$, a change of 4% on a perturbation of one coefficient by less than one part in a billion (see also Problem 9.2.2).

Problems

P9.2.1 The simplest (and least accurate) way to calculate the first derivative of a function is to simply use the definition:

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

Fixing h at some small value, our approximation is

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}.$$

Using the function $f(x) = e^x$, which value of h (to the nearest power of 10) gives the most accurate approximation to $f'(1) = e$?

P9.2.2 Use NumPy's `Polynomial` class (see Section 6.4) to generate an object representing Wilkinson's polynomial from its roots to the available numerical precision; then find the roots of this representation of the polynomial.

9.3 Programming techniques and software development

9.3.1 General remarks

Commenting code

Throughout this book we have tried to comment the code examples and exercise solutions helpfully. This is a good practice, even for short scripts, but the effective use of comments is not an entirely trivial activity. Here is some general advice:

- Generally, prefer to place comments on their own lines rather than "inline" with code (that is, after but on the same line as the code they describe):

  ```
  # Volume of a dodecahedron of side length a
  V = (15 + 7 * np.sqrt(5)) / 4 * a**3
  ```

  rather than

  ```
  V = (15 + 7 * np.sqrt(5)) / 4 * a**3 # Volume of a dodecahedron of side a
  ```

- Explain *why* your code does what it does, don't simply explain *what* it does. Assume that the person reading your code knows the syntax of the language already. Thus,

  ```
  # Increase i by 10:
  i += 10
  ```

  is a terrible comment which adds nothing to the line of code it purports to explain. On the other hand,

  ```
  # Skip the next 10 data points
  i += 10
  ```

  at least gives some indication of the reason for the statement.
- Keep comments up-to-date with the code they explain. It is all too easy to change code without synchronizing the corresponding comments. This can lead to a situation that is worse than having no comment at all:

  ```
  # Skip the next 10 data points
  i += 20
  ```

  Which is correct? Is the comment correct in explaining the programmer's intention but the line of code buggy, or has the line of code been updated for some reason without changing the comment? If your code is likely to be subject to such changes, consider defining a separate variable to hold the change in `i`:

  ```
  DATA_SKIP = 10
  ...
  # Skip the next DATA_SKIP data points
  i += DATA_SKIP
  ```

  In fact, some programmers advocate aiming to minimize the number of comments by carefully choosing meaningful identifier names. For example, if we rename our index, we might even do away with the comment altogether:

  ```
  data_index += DATA_SKIP
  ```

- Explain functions carefully using docstrings. In Python, all functions have an attribute `__doc__` which is set to the docstring provided in the function definition (see Section 2.7.1). A docstring is usually a multiline, triple-quoted string providing an explanation of what the function does, the arguments it takes and the nature of its return value(s), if any. From an interactive shell, typing `help(function_name)` provides more detailed information concerning the function, including this docstring.

Example E9.5 An example of a well-commented function (to calculate the volume of a tetrahedron) is given here.

Listing 9.3 A function to calculate the volume of a tetrahedron

```
# eg9-tetrahedron.py
import numpy as np

def tetrahedron_volume(vertices=None, sides=None):
    """
    Return the volume of the tetrahedron with given vertices or side lengths.
    If vertices are given they must be in an array with shape (4,3): the
    position vectors of the four vertices in three dimensions; if the six sides are
    given, they must be an array of length 6. If both are given, the sides
    will be used in the calculation.

    Raises a ValueError if the vertices do not form a tetrahedron (e.g.,
    because they are coplanar, colinear or coincident).

    """

    # This method implements Tartaglia's formula using the Cayley-Menger
    # determinant:
    #           |0   1    1    1    1  |
    #           |1   0   s1^2 s2^2 s3^2|
    # 288 V^2 = |1  s1^2  0   s4^2 s5^2|
    #           |1  s2^2 s4^2  0   s6^2|
    #           |1  s3^2 s5^2 s6^2  0  |
    # where s1, s2, ..., s6 are the tetrahedron side lengths.

    # Warning: this algorithm has not been tested for numerical stability.

    # The indexes of rows in the vertices array corresponding to all
    # possible pairs of vertices
    vertex_pair_indexes = np.array(((0, 1), (0, 2), (0, 3),
                                    (1, 2), (1, 3), (2, 3)))
    if sides is None:
        # If no sides were provided, work them out from the vertices
❶       vertices = np.asarray(vertices)
        if vertices.shape != (4,3):
            raise TypeError('vertices must be a numpy array with shape (4,3)')
        # Get all the squares of all side lengths from the differences between
        # the 6 different pairs of vertex positions
        vertex1, vertex2 = vertex_pair_indexes.T
        sides_squared = np.sum((vertices[vertex1] - vertices[vertex2])**2,
                               axis=-1)
```

```
else:
    # Check that sides has been provided as a valid array and square it
    sides = np.asarray(sides)
    if sides.shape != (6,):
        raise TypeError('sides must be an array with shape (6,)')
    sides_squared = sides**2

# Set up the Cayley-Menger determinant
M = np.zeros((5,5))
# Fill in the upper triangle of the matrix
M[0,1:] = 1
# The squared-side length elements can be indexed using the vertex
# pair indexes (compare with the determinant illustrated above)
M[tuple(zip(*(vertex_pair_indexes + 1)))] = sides_squared

# The matrix is symmetric, so we can fill in the lower triangle by
# adding the transpose
M = M + M.T

# Calculate the determinant and check it is positive (negative or zero
# values indicate the vertices to not form a tetrahedron).
det = np.linalg.det(M)
if det <= 0:
    raise ValueError('Provided vertices do not form a tetrahedron')
return np.sqrt(det / 288)
```

❶ Using `np.asarray` to convert `vertices` into a NumPy array if it isn't one already enables the function to work with any compatible object (such as a `list` of `lists`).

Style Guide for Python Code

The officially recommended coding conventions for Python are provided by a document known as PEP8 (available at www.python.org/dev/peps/pep-0008/). While it is acknowledged that it isn't always appropriate to follow these conventions all the time, Python programmers generally agree that they maximize the comprehensibility and maintainability of code. The focus is on consistency, readability and in minimizing the probability of hard-to-find typographical errors. Some of the highlights are

- Use *four spaces* per indentation level (and never tabs).[9]
- In assignments, put spaces around the = sign; for example, `a = 10`, not `a=10`.
- Use a maximum of 79 characters per line, where you need to split a line of code over more than one line:
  - favor implicit line continuation inside parentheses over the explicit use of the character, \ (see Section 2.3.1);
  - in arithmetic expressions, break around binary operators so that the new line is *after* the operator;
  - as far as possible, line up code so that expressions within parentheses line up.

[9] A good text editor can be configured to automatically expand tabs to a fixed number of spaces.

For example, the following is considered poor style:

```
lengthy_calculation = margin*margin_px + (border*border_px\
                              + padding*padding_px)
```

and might be better written as

```
lengthy_calculation = (margin*margin_px + (border*border_px +
                                           padding*padding_px))
```

- Separate top-level function and class definitions by two blank lines; within a class, separate them by one blank line.
- Use UTF-8 encoding for your source code (in Python 3 this is the default encoding anyway).
- Avoid wildcard imports (`from foo import *`).
- Separate operators from their operands with single spaces unless operations with different priorities are being combined; for example, write `x = x + 5` but `r2 = x**2 + y**2`.
- Don't use spaces around the `=` in keyword arguments; for example, in function calls use `foo(b=4.5)` not `foo(b = 4.5)`.
- Avoid putting more than one statement on the same line separated by semicolons; for example, instead of `a = 1; b = 2`, write `a, b = 1, 2` (see Section 4.3.1).
- *Functions*, *modules* and *packages* should have short, all-lowercase names. Use underscores in function and module names if necessary, but avoid them in package names.
- *Class names* should be in (upper) CamelCase, also known as CapWords; for example, `AminoAcid`, not `amino_acid`.
- Define *constants*[10] in all-capitals with underscores separating words; for example, `MAX_LINE_LENGTH`.

9.3.2 Editors

While, to some extent, the choice of text editor for writing code is a personal one, most programmers favor one with syntax highlighting and the possibility to define macros to speed up repetitive tasks. Popular choices include:

- Sublime Text, a commercial editor with per-user licensing and a free-evaluation option;
- Vim, a widely used cross-platform keyboard-based editor with a steep learning curve but powerful features. The more basic vi editor is installed on almost all Linux and Unix operating systems;
- Emacs, a popular alternative to Vim;
- Notepad++, a free Windows-only editor;
- SciTE, a fast, lightweight source code editor;
- Atom, another free, open-source, cross-platform editor.

[10] Note that Python doesn't really have constants in the same way that, for example, C does.

Beyond simple editors, there are fully featured integrated development environments (IDEs) that also provide debugging, code-execution, intelligent code-completion and access to operating system services. Here are some of the options available:

- Eclipse with the PyDev plugin, a popular free IDE;
- PyCharm, a cross-platform IDE with commercial and free editions;
- PythonAnywhere, an online Python environment with free and paid-for options (https://www.pythonanywhere.com/);
- Spyder, an open source IDE for scientific programming in Python, which integrates NumPy, SciPy, Matplotlib and IPython.

9.3.3 Version control

Unless properly managed, larger software projects (in practice, anything consisting of more than a single file of code) often rapidly descend into a tangle with modified versions, experimental code, ad hoc features and temporary files. The management of changes to the files comprising a software project is called *version control* (or *revision control*).

At its simplest, version control can involve simply keeping code in a number of parallel directories (folders), numbered chronologically as the software evolves. This approach can work, but if a small change in a large amount of code leads to a new version it is inefficient (a lot of unchanged code is copied across to the new directory). If a new version is created only when the code changes a lot, then there is scope for a lot of tangled code to be generated between versions.

To solve these problems, there are several version control software packages available, some of which are listed here. Most of these run as standalone applications on an operating system and can be invoked from the command line or used through a graphical interface. Some advantages are as follows:

- Many developers can collaborate on one project;
- *Branching*: the parallel development of two versions of the software at the same time, for example, to test out new features;
- *Tagging* (or *labeling*): a way of referring to a snapshot of the project in a particular state;
- Roll-back of a file in the project to a previous version;
- *Cloning*: a means of distributing a software project along with its history of changes;
- Some version control systems integrate with online repositories for storing and sharing code. The most famous of these is GitHub (https://github.com/).

We will not describe the working of version control systems in detail (the syntax varies between systems and there are extensive tutorials, documentation and even entire books written about each one). Some recommended options are:

- Git: the most widely adopted version control system, Git works on a *distributed* (or *decentralized*) basis, allowing developers to work on a project without sharing

a common network or central reference code repository. Open source projects can be hosted for free at *GitHub*.
http://git-scm.com/

- Mercurial: another distributed version control system.
  http://mercurial.selenic.com/
- Subversion (SVN): a centralized option with free (for open source projects) hosting at SourceForge (http://sourceforge.net/). As Git has gained in popularity, SVN is not as widely used as it once was.
  http://subversion.apache.org/

9.3.4 Unit tests

Unit testing is a way of validating software by focusing on individual units of source code. As an object-oriented programming language, for Python this usually means that individual classes (and sometimes even individual functions) are tested against a set of trial data (some of which may be deliberately incorrect or malformed). The aim is to catch any bugs which lead to the faulty interpretation of data. The set of unit tests also serve as a documented and verifiable assertion that the code does what it is supposed to. In some paradigms of code development, unit tests are written before the code itself.[11]

An important advantage of unit testing is that it provides a means of assuring that subsequent changes to the code (perhaps the addition of some functionality) does not break it: the upgraded code should pass the same unit tests that the original code did.

Unit testing your own code for a small project takes discipline. The tests are, themselves, computer code (and, perhaps, associated data) and need careful thought to write. The devising of suitable unit tests often prompts the programmer to think more deeply about the implementation of their code and can catch possible bugs before it is written.

Python's unit testing framework is based around the `unittest` module: a simple application is given in the example.

Example E9.6 Suppose we want to write a function to convert a temperature between the units Fahrenheit, Celsius and Kelvin (identified by the characters `'F'`, `'C'` and `'K'` respectively). The six formulas involved are not difficult to code, but we might wish to handle gracefully a couple of conditions that could arise in the use of this function: a physically unrealizable temperature (< 0 K) or a unit other than `'F'`, `'C'` or `'K'`.

Our function will first convert to Kelvin and then to the units requested; if the from-units and the to-units are the same for some reason, we want to return the original value unchanged. The function `convert_temperature` is defined in the file `temperature_utils.py`.

Listing 9.4 A function for converting between different temperature units

```
# temperature_utils.py

def convert_temperature(value, from_unit, to_unit):
    """ Convert and return the temperature value from from_unit to to_unit. """
```

[11] In particular, so-called 'extreme' programming.

```
    # Dictionary of conversion functions from different units *to* K
    toK = {'K': lambda val: val,
           'C': lambda val: val + 273.15,
           'F': lambda val: (val + 459.67)*5/9,
          }
    # Dictionary of conversion functions *from* K to different units
    fromK = {'K': lambda val: val,
             'C': lambda val: val - 273.15,
             'F': lambda val: val*9/5 - 459.67,
            }

    # First convert the temperature from from_unit to K
    try:
        T = toK[from_unit](value)
    except KeyError:
        raise ValueError('Unrecognized temperature unit: {}'.format(from_unit))

    if T < 0:
        raise ValueError('Invalid temperature: {} {} is less than 0 K'
                                 .format(value, from_unit))

    if from_unit == to_unit:
       # No conversion needed!
        return value

    # Now convert it from K to to_unit and return its value
    try:
        return fromK[to_unit](T)
    except KeyError:
        raise ValueError('Unrecognized temperature unit: {}'.format(to_unit))
```

To use the `unittest` module to conduct unit tests on the `convert_temperature`, we write a new Python script defining a class, `TestTemperatureConversion`, derived from the base `unittest.TestCase` class. This class defines methods that act as tests of the `convert_temperature` function. These test methods should call one of the base class's *assertion functions* to validate that the return value of `convert_temperature` is as expected. For example,

```
self.assertEqual(<returned value>, <expected value>)
```

returns `True` if the two values are exactly equal and `False` otherwise. Other assertion functions exist to check that a specific exception is raised (e.g., by invalid arguments) or that a returned value is `True`, `False`, `None`, and so on. The unit test code for our `convert_temperature` function is here.

Listing 9.5 Unit tests for the temperature conversion function

```
from temperature_utils import convert_temperature
import unittest

class TestTemperatureConversion(unittest.TestCase):

    def test_invalid(self):
        """
        There's no such temperature as -280 C, so convert_temperature should
        raise a ValueError.
```

```
        """
❶       self.assertRaises(ValueError, convert_temperature, -280, 'C', 'F')

    def test_valid(self):
        """ A series of valid temperature conversions to test. """

        test_cases = [((273.16, 'K',), (0.01, 'C')),
                      ((-40, 'C'), (-40, 'F')),
                      ((450, 'F'), (505.3722222222222, 'K'))]

        for test_case in test_cases:
            ((from_val, from_unit), (to_val, to_unit)) = test_case
            result = convert_temperature(from_val, from_unit, to_unit)
❷           self.assertAlmostEqual(to_val, result)

    def test_no_conversion(self):
        """
        Ensure that if the from-units and to-units are the same the
        temperature is returned exactly as it was passed and not converted
        to and from Kelvin, which may cause loss of precision.

        """
        T = 56.67
        result = convert_temperature(T, 'C', 'C')
❸       self.assertEqual(result, T)

    def test_bad_units(self):
        """ Check that ValueError is raised if invalid units are passed. """
        self.assertRaises(ValueError, convert_temperature, 0, 'C', 'R')
        self.assertRaises(ValueError, convert_temperature, 0, 'N', 'K')

unittest.main()
```

❶ `assertRaises` verifies that a specified exception is raised by the method `convert_temperature`. The necessary arguments to this method are passed after the method object itself.

❷ We need `assertAlmostEqual` here because the floating point arithmetic is likely to cause a loss of precision due to rounding errors.

❸ We use `assertEqual` here to ensure that the temperature value is returned as it was passed and not converted to and from Kelvin.

Running this script shows that our function passes its unit tests:

```
$ python eg9-temperature-conversion-unittest.py
...
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

9.3.5 Further Reading

- F. Brooks, (1975, 1995). *The Mythical Man-Month*, Addison-Wesley. Near-legendary monograph on software development explaining why "adding man-power to a late software project makes it later."
- J. Loeliger and M. McCullough, (2012). *Version Control with Git*, O'Reilly.
- S. McConnell, (2004). *Code Complete: A Practical Handbook of Software Construction*, Microsoft Press.
- A. Hunt and D. Thomas, (1999). *The Pragmatic Programmer*, Addison-Wesley.