# High-Performance Computing: Numerics

Adrian F. Clark: alien@essex.ac.uk

2016–17

# The fundamentals

Computers store numbers in ways that pack them into a fixed number of *bits* (binary digits) or, more commonly, *bytes* (groups of 8 bits)

The two most widely-used data types are *fixed-point* and *floating-point* numbers

All other data types (characters, dictionaries, complex numbers, *etc*) can be built on top of these

# Integers

In principle, any whole number can be represented as an integer, though the number of bits restricts this in practice

The representation needs enough bits to store all values in computations, including intermediate values

```
unsigned char i1, i2=255;
i1 = i2 + 1;
i1 = (i2 + 2) / 2;
i1 = i2 / 2 + 1;
```

Integer *overflow* (where the value is too large to be stored in the available number of bits) and *underflow* (the converse) are usually **not reported**

# Integer division

Any remainder from a division operation is discarded

```
int i1, i2=35, i3=6;
i1 = i2 / i3;
```

In numerical computing, this is often not what one wants; it is better if the number is rounded correctly so that a fraction $\geq 0.5$ is rounded up to the next integer

```
int i1=6, i2=35, i3, i4, i5;
float f3;
f3 = (float)i2 / i1;
i3 = (int)(f3 + 0.5);
```

The usual way people round can cause overflow

```
i3 = (i2 + i1 - 1) / i1;
```

A better solution is

```
i4 = (i2 - 1) / i1 + 1;
```

However, neither of these round correctly

```
int main (int argc, char *argv[])
{
  int i1 = 6, i2, i3, i4, i5;

  printf ("      i3  i4  i5\n");
  for (i2 = 30; i2 <= 36; i2++) {
    i3 = (i2 + i1 - 1) / i1;
    i4 = (i2 - 1) / i1 + 1;
    i5 = (i2 + i1/2) / i1;
    printf ("%3d: %3d %3d %3d\n", i2, i3, i4, i5);
  }
  return EXIT_SUCCESS;
}
```

However, even this works only when both `i1` and `i2` are positive

# When integer arithmetic gives out

Consider how far you live from the University. It is unlikely to be a whole number of miles, km or metres

If you need to work out how long light would take to travel between your home and a point at the University, you would need to divide that distance by the speed of light (299 792 458 m/s)

The result is under one second, so representing it as an integer is impractical; this is where floating-point numbers come in

# Scientific notation

A number such as 314.15 is written as $3.1415 \times 10^2$ or $31.415 \times 10^1$

When there is a single non-zero digit to the left of the decimal point, we say the number is *normalized*

How does one add two numbers expressed in scientific notation?

# Floating-point numbers

Floating-point numbers are the computer analogue of scientific notation, but using binary rather than decimal digits

A number is represented by a *sign bit* $s$, an integer *exponent* $e$ and a positive integer *mantissa* $M$:

$$s \times M \times B^{e-E}$$

$B$ is the *base* of the representation, 2, and $E$ is the *bias* of the exponent, a fixed integer for any given computer and representation

It is possible to write programs to work out important properties of floating-point arithmetic

# IEEE-format floating-point numbers

There is a standard, IEEE 754, for representing floating-point numbers, and practically all computers built since about 1980 use it

In *single precision* and *double precision* numbers (`float` and `doub;e` in C), the layouts are

| locations | size | meaning |
|---|---|---|
| 31 | 1 bit | sign $s$ |
| 30–23 | 8 bits | exponent $e$ |
| 22–0 | 23 bits | mantissa $m$ |
| 63 | 1 bit | sign $s$ |
| 62–52 | 11 bits | exponent $e$ |
| 51–0 | 52 bits | mantissa $m$ |

With `float`, the range of numbers that can be represented is roughly

$$\pm 10^{-38} \ldots 10^{38}$$

while with `double` it is roughly

$$\pm 10^{-308} \ldots 10^{308}$$

For comparison, there are thought to be about $10^{80}$ atoms in the universe.

An arbitrary number may not be able to be represented exactly: this will obviously be the case for irrational numbers such as $\pi$, $e$ and $\sqrt{2}$ but numbers such as 3 or any number with many decimal places will also exceed the representation

Arithmetic between floating-point numbers is inexact, even if the operands are represented exactly

Consider adding two floating-point numbers: this proceeds by successively right-shifting (dividing by 2) the mantissa of the smaller operand, simultaneously incrementing its exponent, until both operands have the same exponent

The mantissas are then added and the result normalized

This right-shifting of the mantissa of the smaller operand means that its least significant digits are discarded — and if the two operands differ by too much, the smaller is effectively replaced by zero

# Round-off error

The smallest (in terms of magnitude) floating-point number which, when added to 1.0, produces a value different from 1.0 is termed the *machine accuracy* and is often denoted as $\epsilon_M$. A typical 32-bit floating-point number with $B = 2$ has $\epsilon_M \approx 3 \times 10^{-8}$

Any floating-point operation should be expected to introduce an additional fractional error of at least $\epsilon_M$; this is known as *round-off error* and is an inherent property of floating-point arithmetic

Round-off errors accumulate as the amount of computation increases; for $N$ operations, the best we can hope is that the total round-off error will be $\sqrt{N}\epsilon_M$ — but it is common for it to be $N\epsilon_M$

# When round-off becomes dangerous

There are two cases when round-off becomes dangerous:

- when (*e.g.*,) the two operands of addition or subtraction differ by enough orders of magnitude that one is "shifted out"
- when two numbers of almost equal magnitude are subtracted, so that only the low-order bits of the mantissa differ — the most significant bits of the result will then be affected by round-off error

It is important to remember that round-off error is a characteristic of floating-point arithmetic and little can be done about it on a given computer

IEEE-format floating-point arithmetic ties down the way rounding is done — and people who understand numerical programming sometimes include special code to reduce round-off for IEEE-compliant arithmetic

# Truncation error

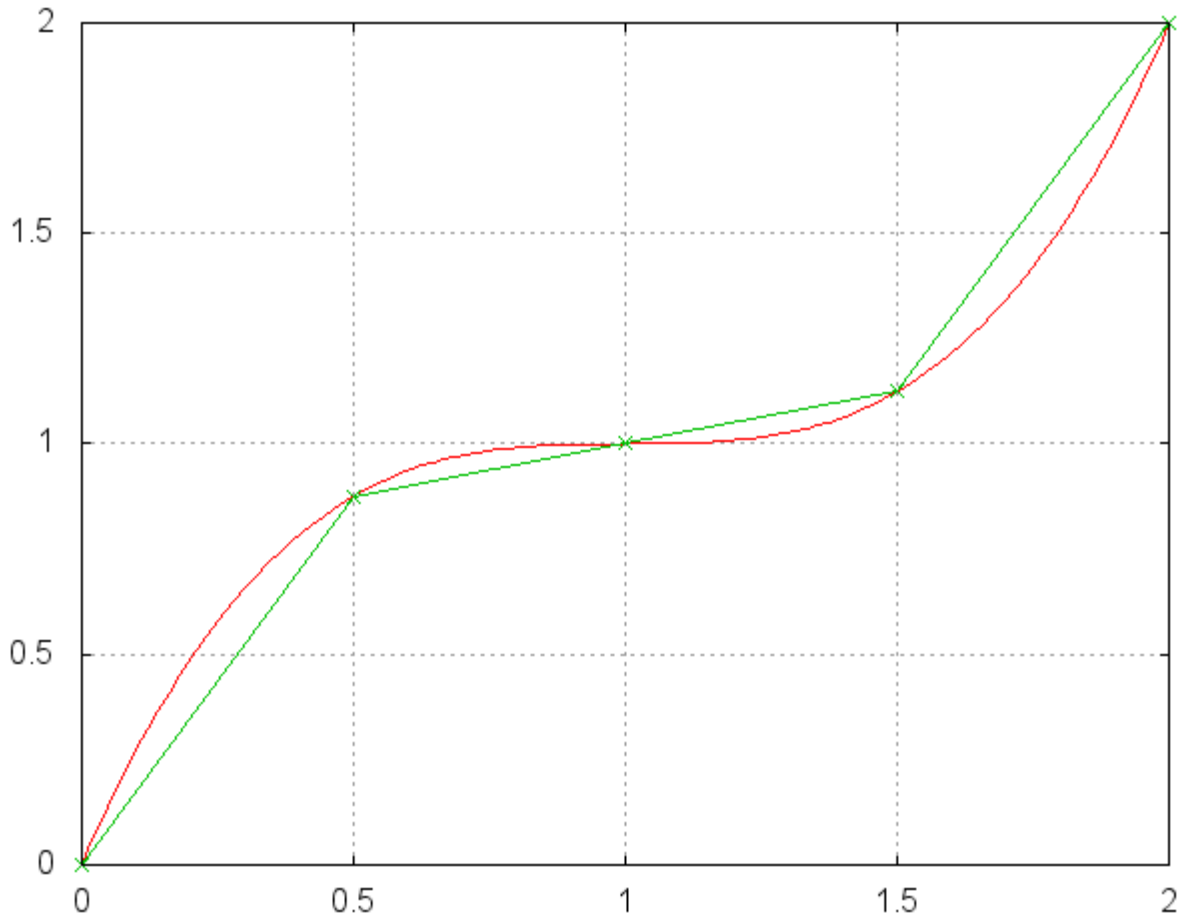


Figure 1:Cubic curve and its approximation

Consider calculating the integral (the area under the curve) of the red line in the graph

One way is to split the curve into a series of steps, allowing us to add up the areas of squares and triangles as shown in the green line

Here, for $x < 1$ this approach *underestimates* the true value, while for $x > 1$ it *overestimates* it

This discrepancy between the true result and the numerical one is called *truncation error* and is quite different from round-off error — it would exist even if there were no round-off error or the calculation was done using integers

Truncation error *is* under the control of the programmer

In this case, we could decrease the sampling step size; but there are also better integration algorithms than the one shown here

# Numerical Methods

# Use a library or roll your own?

I used to recommend that people find a good algorithm for any numerical computation and write their own code

Nowadays, it makes more sense to look for off-the-shelf numerical software — LINPACK, LAPACK, EIGEN, fftw, and many others — as the algorithms are well-thought-out and robust

The best numerical software is the (commercial) NAg library but that is becoming increasingly difficult to get hold of

Whatever software you use, do ensure it gives sensible results on data typical of *your* problem

# Calculating the standard deviation

The s.d., $\sigma$, of a set of $N$ numbers is calculated from

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2$$

where $\bar{x}$ is the mean — this needs two passes through the data to calculate

We can expand the square and re-formulate this as

$$\sum x^2 - \frac{(\sum x)^2}{N}$$

which needs only one pass through the data

Usually this works fine — but if the values are large and the amount of variation small, we run into the problem of subtracting almost equal numbers alluded to above, and incorrect results often ensue

# Solution of a quadratic

We are taught at school that the solution to $ax^2 + bx + c = 0$ is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

When the discriminant, $b^2 - 4ac$, involves values that make $b^2 \gg 4ac$ then $4ac \to 0$ relative to $b^2$ so that the discriminant becomes $\pm b \ldots$ and this means that the lower of the two solutions to the equation is $-b + b = 0$

Numerical analysts devised mathematically-equivalent formulations that are more stable

We first calculate

$$q = -\frac{1}{2}\left(b + \mathrm{sgn}(b)\sqrt{b^2 - 4ac}\right)$$

then the two solutions to the quadratic are given by

$$x_1 = c/q$$

and

$$x_2 = q/a$$

# Solving sets of equation

How would you solve these simultaneous equations?

$$2x + 3y = 9$$

$$3x + 2y = 8.5$$

Consider the case when there are $N$ equations in $M$ unknowns

$$A_{11}X_1 + A_{12}X_2 + \cdots + A_{1M}X_M = B_1$$
$$A_{21}X_1 + A_{22}X_2 + \cdots + A_{2M}X_M = B_2$$
$$\cdots$$
$$A_{N1}X_1 + A_{N2}X_2 + \cdots + A_{NM}X_M = B_N$$

If $N \geq M$ (the "over-determined" case), we can find a solution; but if $N < M$ ("under-determined"), we can only minimize the overall error

# Gaussian elimination

We manipulate one of the equations so that subtracting it from one of the others makes the coefficient one $X_i = 0$

We repeat this process until one equation in one unknown is left

We back-substitute the result into the equation for two unknowns and solve it, and so on until we have solved all the equations

Let us now think about the numerical accuracy of the approach

As Gaussian elimination involves scaling one equation and subtracting it from another, we are again susceptible to round-off errors

To ameliorate this, in each iteration we can choose the $A_{ij}$ with the largest magnitude

However, there are better algorithms than Gaussian elimination; *e.g.*, singular value decomposition (SVD) is the best linear least-squares solution in both the under- and over-determined cases

# Matrix multiplication

High-performance computing generally involves large numbers of calculations or a lot or data (or both)

For many problems, we end up with large (say $> 20,000 \times 20,000$ element) matrices of numbers that have to be multiplied together

In computer graphics, vast numbers of $4 \times 4$-element matrix multiplications have to be performed in real time

Let us write

- $\mathbf{A}$ is a $N \times L$ matrix
- $\mathbf{B}$ is a $L \times M$ matrix

so that $\mathbf{C} = \mathbf{AB}$ is a $N \times M$ matrix

Each element of **C** is calculated as

$$C_{i,j} = \sum_{k=0}^{L} A_{i,k} B_{k,j}$$

This translates into code as

```
for (i = 0; i < n; i++)
  for (j = 0; j < m; j++)
    C[i][j] = 0.0;
    for (k = 0; k < L; k++)
      C[i][j] += A[i][k] * b[k][j];
```

which clearly scales as $O(n^3)$ — for each element in **C**, we do $L$ multiplications and additions

Can we speed this up in the general case? What about for $4 \times 4$ matrices?