Advanced Algorithms II

Todd Davies

February 6, 2016

Overview

This course will explore certain classes of algorithms for modelling and analysing complex systems, as arising in nature and engineering. These examples include: flocking algorithms - e.g., how schools of fish or flocks of birds synchronised; optimisation algorithms; stability and accuracy in numerical algorithms.

Aims

By the end of the course, students should:

- Appreciate the role of using nature-inspired algorithms in computationally hard problems.
- Be able to apply what they learnt across different disciplines.
- Appreciate the emergence of complex behaviours in networks not present in the individual network elements.

Syllabus

Attribution

These notes are based off of both the course notes (found on Blackboard). Thanks to the course staff (¡names¿) for such a good course! If you find any errors, then I'd love to hear about them!

Contribution

Pull requests are very welcome: `https://github.com/Todd-Davies/third-year-notes`

Contents

1 Finite precision computation

Unfortunately, the world is not solely restricted to integers, and computers often need to work with real numbers $\mathbb{R}$. With integers, the main problem we have in computer terms is overflow, and since there is a finite distance from one to the next, they are easy to encode in a computer.

On the other hand, between any two real numbers, there are infinitely many more real numbers. Since computers are discrete, we need to sample the real numbers so that we can find a representation for them in the computer. However, this introduces errors, since we can't represent every value exactly and therefore most approximate.

1.1 Floating point numbers

One problem we have with computation is that we don't know what the error is with computations; how 'good' is the result of an algorithm or computation? We would like to know the error bounds of a solution, and have the output be reliable.

In the 70's, it was realised that different floating point implementations produced different results. This had significant concerns for reproducability, and as a result the ANSII IEEE standard for binary floating point arithmetic was created.

Each floating point number is represented as four integers; the base, the precision, the exponent and the mantissa.

$$x = \pm m \times b^{e-n}$$

Where:

m	The Mantissa (the bit before the decimal place)
b	The base (or radix), usually two or ten
e	The exponent (the power of the radix)
n	The precision (the numbewr of digits in the mantissa)

We can represent different numbers in different ways, for example:

$$0.121e10^3 = 0.0121e10^4$$

In this case, we can normalise the way in which we represent numbers and at least all computers will get the same errors.

The amount of numbers we can represent with the floating points depends on the values permissable for b, n and e. When $b = 2, n = 2$ and $e = [-2 \ldots 2]$:

The Python code used to generate this is found in the /COMP36212/programs folder of the source for these notes.

$$\begin{array}{ccccc}
2 & \times & 2^{-2} & = & 0.500000 \\
3 & \times & 2^{-2} & = & 0.750000 \\
2 & \times & 2^{-1} & = & 1.000000 \\
3 & \times & 2^{-1} & = & 1.500000 \\
2 & \times & 2^{0} & = & 2.000000 \\
3 & \times & 2^{0} & = & 3.000000 \\
2 & \times & 2^{1} & = & 4.000000 \\
3 & \times & 2^{1} & = & 6.000000 \\
2 & \times & 2^{2} & = & 8.000000 \\
3 & \times & 2^{2} & = & 12.000000
\end{array}$$

The mantissa is always 2 or 3 since we're using an explicit one, so the binary values are either $[1, 0]$ or $[1, 1]$.

Floating point numbers are relatively spaced; even though they might not be the same distance apart, the ratio between them is the same. The unit round off (basically the last digit) is called the *relative machine precision*.

The **Relative Machine Precision** is given by $u = 0.5 \times b^{1-n}$, and is the largest possible difference between a real number and its floating point representation. In the above example, $u = 0.5 \times 2^{-1} = \frac{1}{4}$. The value $2u$ is called the **Machine Precision**.

In the exam, assume explicit storage of leading bit of mantissa.

1.1.1 Real world floats

For the sign bit, 0 is for positive numbers and 1 is for negative ones. The exponent must also be able to represent negative numbers (in the case of 24×2^{-2} for example), and thus in single precision floats, a bias of +127 is added to the exponent and that value is stored. The exponent values −127 and +128 are reserved for special numbers.

The first bit of the mantissa is implicitly 1 in the IEEE base two floating point representation. This is because normalised numbers always have 1 as the first digit of their mantissa, and then we can get another digit of precision.

Sixty-four bit floating point numbers have one sign bit, 11 exponent bits and 52 mantissa bits. This means their bias will be $2^{1}1 = 2048$ and the range will be $2 - 2^{52} \times 2^{2^{11}}$.

The standard also has special values built in:

Zero: When the exponent is all zeros and the mantissa equal to zero.

Denormalised number: If the exponent is all zero, but the mantissa is non-zero, then the number is $-1^{sign} \times 0.m \times 2^{-126}$.

Infinity: Exponent is all 1's and mantissa is all 0's. The sign dictates between positive and negative inifinity.

NaN: Not your grandma, this is when the value isn't a real number, such as when a dividion by zero occurs.The exponent is all 1's and the mantissa is non-zero.

1.1.2 Error in floating point numbers

When a real number is converted to floating point number, it may lose precision. If the real number if x and the floating point representation is $\bar{x}$, then the error is:

$$e = \bar{x} - x$$

You can find how many significant digits a floating point number approximates a real number to by doing:

$$|\bar{x} - x| = 10^{-\text{significant digits}}$$

However, the absolute error e does not give us a very good description of the accuracy (if the error is 10^{-6} but the value is 10^{-7} then we're very, very inaccurate)! To rectify this, we have relative error:

$$r = \frac{e}{x} = \frac{\bar{x} - x}{x}$$

When we're getting the floating point number from the real number, we can trucate the (possibly infinate) digts so that it fits in the mantissa. Simply chopping the number so it fits in m bits is called **simple trucation**.

If we round numbers instead of using simple truncation, then we can reduce the error. We need some rules though:

- If the part of the mantissa to be chopped of is less than 0.5, use simple truncation.

- If it's greater, then increment the last digit of the mantissa, and then truncate.
- If it's equal to 0.5, then we can do either (though IEEE says to round up).

Now the relative error is:

$$|r| = \frac{|e|}{|x|} = \frac{0.5 \times b^{e-n}}{|m| \times b} = \frac{1}{2 \times m \times b^n}$$

There are three types of errors that computers can make:

- Essential errors are ones that cannot be avoided (e.g. from erroneous input).
- Rounding errors are when we have to approximate real numbers with floating point ones. This error can be measured and controlled.
- Methodology errors come into play when replacing one problem by another similar, easier but less accurate problem is done. The solution is close, but not exact.

Unfortunately, errors can propogate through a computation. We must know the errors introduced by every operation a computer performs on floating point numbers. If we know $e_x = \bar{x} - x$ and $e_y = \bar{y} - y$, what is $e_{x \cdot y}$?

The error introduced by addition, subtraction, multiplication and division is:

Addition:

$$\bar{x} + \bar{y} = (x + e_x) + (y + e_y) = (e_x + e_y) + (x + y)$$

$$e_{x+y} = e_x + e_y$$

Subtraction:

$$e_{x-y} = e_x - e_y$$

Multiplication:

$$\bar{x} \times \bar{y} = (x + e_x) \times (y + e_y) = xy + xe_y + ye_x + e_x e_y$$

$$e_{x \times y} \equiv xe_y + ye_x$$

Division:

$$e_{\frac{x}{y}} \equiv \frac{1}{y} e_x - \frac{x}{y^2} e_y$$