Advanced Algorithms II

Todd Davies

February 21, 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-Davi` `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 number 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 permissible for b, n and e. When $b = 2, n = 2$ and $e = [-2 \ldots 2]$:

Note:

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

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

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 $2^11 = 2048$ 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 infinity.

NaN: Not your grandma, this is when the value isn't a real number, such as when a division 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 truncate the (possibly infinite) digits so that it fits in the mantissa. Simply chopping the number so it fits in m bits is called **simple truncation**.

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 propagate 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:

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

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

Division:

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

Relative error can also be calculated:

Addition:

$$r_{x+y} = \frac{e_{x+y}}{x+y} = \frac{x}{x+y} r_x + \frac{y}{x+y} r_y$$

Subtraction:

$$r_{x-y} = \frac{e_{x-y}}{x-y} = \frac{x}{x-y} r_x + \frac{y}{x-y} r_y$$

Multiplication:

$$r_{x \times y} = \frac{e_{x \times y}}{x \times y} \approx r_x + r_y$$

Division:

$$r_{x/y} = \frac{e_{x/y}}{x/y} \approx r_x - r_y$$

In general, $\bar{x} \circ \bar{y} = (x \circ y)(1 + r_{x \circ y})$.

Note:

Remember, error propagation is not associative. The error from a multiplication and then an add is probably not the same as doing the add then the multiplication.

While it is useful to know the error of one operation, we also need to be able to work out the error of consecutive operations. That is to say given $e_x = \bar{x} - x$ and $e_y = \bar{y} - y$, determine $e_{x \bar{\circ} y}$.

$$\begin{aligned} \bar{x} &= FP(\bar{x_1} \circ \bar{x_2}) \\ &= \bar{x_1} \bar{\circ} \bar{x_2} \\ &= (\bar{x_1} \circ \bar{x_2})(1+u) \end{aligned}$$

Where $|u| \leq 0.5 \times b^{-n+1}$.

The total relative error is given by:

$$r_z^t = \frac{e_z^t}{x} = r_{x \circ y} + u = a_x r_x + a_y r_y + u$$

a_x and a_y is the error introduced by the $\circ$ operation.

Lets put all that into an example. Given the numbers x, y and x with their relative round off errors r_x, r_y and r_z, determine the relative error in $u = (x+y)z$:

$$r_{x+y}^t = \frac{x}{x+y} r_x + \frac{y}{x+y} r_y + r_+$$

$$r_u^t = \frac{x}{x+y} r_x + \frac{y}{x+y} r_y + r_+ + r_z + r_*$$

We can work out the unit round off errors somehow...

1.2 Finding the number of significant digits

We want to find an integer that represents how many digits in our number are non-nonsense (i.e. how many significant digits we have). The number of significant digits in the floating point number $\bar{x}$ where its real equivalent is x is:

Note:

Z is the set of integers.

$$l = Z(log_b \frac{|x|}{|\bar{x} - x|})$$

If we rearrange this, the relative error is:

$$r_x \approx b^{-l}$$

If we have a computation that takes m real numbers as arguments and outputs a real number, if the arguments are floating point numbers with l_i significant digits then we can estimate:

$$|e| \approx |\sum_{i=1}^{m} x_i \frac{\delta f}{\delta x_i} b^{-l_i}|$$

Also:

$$|\sum_{i=1}^{m} x_i \frac{\delta f}{\delta x_i} b^{-l_i}| \leq b^{-l_{min}} |\sum_{i=1}^{m} x_i \frac{\delta f}{\delta x_i}|$$

The number of significant digits in the answer is:

$$l = l_{min} - \delta$$

Where δ is the loss of significant digits:

$$\delta = Z(log_b(\frac{\sum_{i=1}^{m} |x_i \frac{\delta f}{\delta x_i}|}{|f(x_1, \ldots, x_m)|})$$

If we try and subtract numbers that are close in magnitude, then we will lose lots of significant digits. If we do $\sqrt{2.01} - \sqrt{2}$ (where both numbers are known to 9 significant digits), then we get:

$$\delta = Z(log_1 0(\frac{|\sqrt{2.01}| + |\sqrt{2}|}{|\sqrt{2.01} - \sqrt{2}|})) = 3$$

Our answer would be to six significant figures. In order to get all of the significant figures, we need to use a different method:

$$\begin{aligned} z &= \sqrt{2.01} - \sqrt{2} \\ &= (\sqrt{2.01} - \sqrt{2})\frac{\sqrt{2.01} + \sqrt{2}}{\sqrt{2.01} + \sqrt{2}} \\ &= \frac{\sqrt{2.01}^2 - \sqrt{2}^2}{\sqrt{2.01} + \sqrt{2}} \\ &= \frac{0.01}{\sqrt{2.01} + \sqrt{2}} \end{aligned}$$

Note:

Revision break? Read this: https://randomascii.wordpress.com /2014/01/27/theres-only-four -billion-floatsso-test-them-all/

1.2.1 Accurately computing sample variance

Computing the sample variance of a set of numbers is done by the formula:

$$s_n^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \hat{x})^2$$

Where $\hat{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ is the mean of the values.

In order to compute the variance, we would need to calculate the mean first, then calculate the variance, which requires two for loops. However, a numerically equivalent formula exists for doing the same thing:

$$s_n^2 = \frac{1}{n-1}(\sum_{i=1}^{n} x_i^2 - \frac{1}{n}(\sum_{i=1}^{n} x_i)^2)$$

If we use the following data:

Value	Floating point representation
100	0.1000×10^3
101	0.1010×10^3
102	0.1020×10^3

If we use formula one for the variance with these input numbers, then we get an answer of 1, if we use formula two, then we get the answer to be -1.667

1.2.2 Overflow and underflow

We must be aware of overflow and underflow in our operations. For example, if we were calculating the length of the hypotenuse of a triangle ($\sqrt{\text{opposite}^2 + \text{adjacent}^2}$), then we could have an overflow if $x = y = 10^{200}$, since the range representable by a single precision float is $10^{\pm 308}$.

This can be remedied by using a different formula:

$$a = \max(x, y), \quad b = \min(x, y)$$

$$z = \begin{cases} a\sqrt{1 + \left(\frac{b}{a}\right)^2} & a > 0 \\ 0 & a = 0 \end{cases}$$

1.3 Condition of a problem

The condition of a problem is how sensitive it is to changes in the data. A problem is ill-conditioned if a small change in the data results in a large change in the solution. This is concerned with the problem, not the method used to solve it.

Consider the equation $x^3 - 21x^2 + 120x - 100 = 0$; the solutions are $x = \{1, 10\}$. However, if we change the value of x^3 to 0.99 or 1.01, then the roots become $x = \{1, 11.17, 9.041\}$ and $x = \{1, 9.896 \pm 1.044\}$ respectively.

We can work out the sensitivity by using partial differentiation, if the perturbed function if $\bar{f}(x) = f(x) + \epsilon g(x)$, where $g(x)$ is the change applied to $f(x)$, then:

$$|\delta| \approx |\frac{\epsilon g(x)}{f'(x)}| \text{ (single root)}$$

$$|\delta| \approx |\sqrt{-\frac{2\epsilon g(x)}{f''(x)}}| \text{ (double root)}$$

In the above example, $g(x) = x^3, \epsilon = -0.01, f'(x) = 3x^2 - 42x + 120$:

$$|\delta| \approx |\frac{-0.01 \times 1^3}{81}| \approx 0.0001$$

For the double root, $f''(x) = 6x - 42$

$$|\delta| \approx |\sqrt{-\frac{2 \times -0.01 \times 1000}{18}}| \approx 1.054$$

So the double root is vastly affected, but the single root isn't.

1.4 Stability

The stability of a method is how sensitive it is to rounding errors. A method that guarantees as accurate solution as the input data allows is said to be stable, otherwise it's unstable. Condition is about the sensitivity of the problem, but stability is about the sensitivity of the method.

Given the quadratic equation $1.6x^2 - 100.1x + 1.251 = 0$, the solution can be found using the standard formula (in floating point) ($x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$), which gives $x = \{62.53, 0.03125\}$.

If we only compute the first root to be $x_1 = 62.53$, but use $x = c/ax_1$ for the second, then we get the second root to be 0.01251. The correct solutions are $x = \{62.53, 0.0125\}$.

The problem was that $y_1 = -b = 100.1$, and $y_2 = \sqrt{b^2 - 4ac} = 100.06$. Since if we do subtraction with very close numbers, the error is high. If we calculate δ for this then we get:

$$\delta = X(log_1 0 \frac{|y_1| + |y_2|}{|y_1 - y_2|}) = 4$$

Meaning all of our digits were rubbish!

2 Optimisation and nature inspired algorithms

To start, lets recap some stuff you should already know; a minima is a point of a function where all points in its vicinity have a higher value than itself,and a maxima is the opposite; a point where all points in its vicinity have a lower value than itself.

Since we can invert a function by putting a minus in front of it, we don't really need to differentiate between maxima and minima, since we can always express on in terms of the other.

Global optimums are the best values for the entire domain, while local optimums are best in some bounded region.

One dimensional functions are easiest to visualise; they are just a graph, where the y value is the value of the function and the x value is the input. As the dimensionality increases, the functions get harder to visualise; 2d functions are visualised on a 2d graph, where the intensity of the colour in each square represents the value of the function at that coordinate.

An optimisation problem is one where we attempt to maximise or minimise a function. This involves finding the input that will produce the largest or smallest value. Optimisation problems are really search problems; given a domain, find an input that produces the smallest output.

If we know exactly what the function we're trying to optimise is (i.e. we have an equation), then it's an explicit problem, but if we just have some inputs and outputs, then it's a black box problem.

Similarly, there are two different approaches to solving these problems:

Single solution:
: This is where there is a single candidate solution that is incrementally improved by the algorithm throughout the procedure.

Population based solution:
: This is where there is a set of candidate solutions (the population) and an iterative operation combines the best ones to improve the quality of the population.

All of the optimisation methods follow a cycle:

- Guess some parameters for the initial solution
- While we're not satisfied:
  - Evaluate how the current parameters perform
  - If we're satisfied, then we've finished
  - Otherwise, then determine new parameters based on the current ones and their evaluation.

We can use derivatives to try and find maxima and minima in a function. If we do not have an explicit function for the derivative, then we can calculate them using **finite differences**:

Forward difference:

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

Central difference:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Here, the h parameter is the range over which we're calculating the differential.

2.1 Minimising univariate functions

For one dimensional functions, we can use two approaches:

- Interval reduction, where we iteratively reduce the range of values that we think the optimum is inside.
- Interpolation, where we try to find an approximate function and use the optimum of that function to iterate to find a better approximate function.

2.1.1 Bisection algorithm

This is the easiest algorithm, we start with the full range in the domain, and then cut it in half based on the value of the differential at the midpoint at the range.

```
bisection(f'(x), a, b, e) {
  while (b - a >= e) {
    c = (a + b) / 2;
    if (f'(c) < 0) a = c;
    else if(f'(c) > 0) b = c
    else return c;
  }
}
```

2.1.2 Quadratic interpolation algorithm

Here, we make a quadratic function that goes through three points a, b and c, then we either drop a or b depending on which way we should move the quadratic function. Here, **we do not need the differential** which is good since we won't always have it!

```
Sudo insert psudo code
```

2.1.3 Stopping criteria

Eventually, we will have to stop our optimisation algorithms, but some of them will run indefinitely. We need criteria to make them stop after a sane amount of time. The simple approach is to stop after n iterations, and is applicable to all iterative functions. The more clever approach is to only stop when we reach a certain threshold, i.e. when a method converges asymptotically on an optimum.

2.2 Minimisation of multivariate functions

There are two different classes of minimisation methods for multivariate functions:

Based on derivatives:
Moves are determined based on information from derivatives of the multivariate function.

A *directional derivative* is a derivative indicating the rate of change in a specific dimension:

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

Where r is the direction. If Φ'_r is positive, then the direction is ascending, if it's negative, then the direction is descending and if it's equal to zero, then there is no slope.

Finding the gradient of a vector of a multidimensional function $f(x)$ (and hence the vector of all partial derivatives) is:

$$\frac{\delta f}{\delta x_j} = lim_{h \to 0} \frac{f(x + he_j) - f(x)}{h}$$

e_j is the unit vector in the direction of axis j, so to construct the whole gradient for the vector, we need to do:

$$\Delta f(x) = \begin{bmatrix} \frac{\delta f}{\delta x_1}(x) & \frac{\delta f}{\delta x_2}(x) & \ldots, \frac{\delta f}{\delta x_n}(x) \end{bmatrix}^T$$

The **Hessian** is the matrix containing the second order partial derivatives, and is always of size $N \times N$:

$$Hf(x) = \begin{bmatrix} \frac{\delta^2 f}{\delta x_1 \delta x_1}(x) & \ldots & \frac{\delta^2 f}{\delta x_1 \delta x_n}(x) \\ \vdots & \ddots & \vdots \\ \frac{\delta^2 f}{\delta x_n \delta x_1}(x) & \ldots & \frac{\delta^2 f}{\delta x_n \delta x_n}(x) \end{bmatrix}$$

Most algorithms have some kind of *parameter update scheme*, where they will move in a given direction r for a length of α every iteration, and r, α change based on the current position:

$$x_{i+1} = x_i + r_i \alpha_i$$

2.2.1 The alternating variable method

A conceptually easy method, you start from an arbitrary position x, and then for each dimension d:

- Find the differential for that dimension
- Find the minimum according to that differential
- Move the value of x_d to be equal to that minimum.

Then repeat for all dimensions until no progress is made.

2.2.2 Newton's method

As soon as Newton invented calculus, he also started inventing methods for finding minima. This method requires a function to be twice differentiable:

$$x_{i+1} = x_i - \alpha \frac{\Delta f(x_i)}{Hf(x_i))}$$

The trouble with Newton's method is that it will only converge if the initial point is close to the solution, and the memory requirements for the Hessian matrix is n^2.

Direct search:

Moves are determined using methods other than derivatives, for example, using geometric concepts.

2.3 Simplex methods

If you did Advanced Algorithms 1, you're either groaning now or you're jumping for joy at being able to skip a section. However, these aren't the simplex methods you learned about last semester.

For this, you construct a simplex (a shape of $n + 1$ vertices in n dimensions), and then observe that you can always form a new simplex from an existing one by adding a new point.

The **Spendley, Hext and Himsworth method** is like so:

1. Create $n + 1$ vectors for a regular simplex (where all sides are the same length).
2. For each vector, the value of the objected point is calculated for that point.
3. The vector with the highest point is removed and substituted by a new one.
4. If the worst vector is also the most recently introduced one, then the next worst one is used.
5. The newly substituted vector is the mirror image of the old one along the axis of the remaining vectors.

6. When the simplex simply rotates around a point, then the vertex in the middle is the minimum.

The maximum age of a vector is $1.65n + 0.05n^2$, and if a vertex exceeds age a, then the search stops, or is restarted using a smaller simplex. If the search was restarted, then the initial (smaller) simplex is started from the site of the rotating.

The **Nelder and Mead method** is similar, though the simplices are no longer regular (though they can be). It involves having rules that either expand, contract or reflect the simplex (or any combination of them), which leads to quicker results and better approximations of the solution.

Where $y_0^{(r)}$ is the best value in the previous simplex, and $y_{n-1}^{(r)}$ is the worst, the new simplex results from:

$y_0^{(r)} < y_i^{(r+1)} < y_{n-1}^{(r)}$:
The simplex is reflected like in the SHH method.

$y_i^{(r+1)} < y_0^{(r)}$:
The simplex is reflected and expanded.

$y_i^{(r+1)} > y_{n-1}^{(r)}$:
The simplex is reflected and contracted.