Numerical Analysis

Dr Bogdan Roman*

With contributions from:
Daniel Bates, Mario Cekic, Richie Yeung

Computer Laboratory, University of Cambridge

`http://www.cl.cam.ac.uk/teaching/current/NumAnalys`
(The online slide pack contains further additions/corrections)

`v20190529.1`

Course outline

UNIVERSITY OF CAMBRIDGE

The course will *touch* on:

Errors. Bloody errors ...

Numerical calculus. Calculus when we don't know the function?

Iterative methods. Things that converge ... hopefully.

Linear systems. Getting machines to solve (large) systems of equations ...

Data analysis. Can we make sense of 17-dimensional data?

(FDTD. Ever wondered about those realistic physics in computer games?)

Number representation and floating point computation. Teach computers to represent and deal with numbers ... and ourselves to deal with the fallout.

Errors

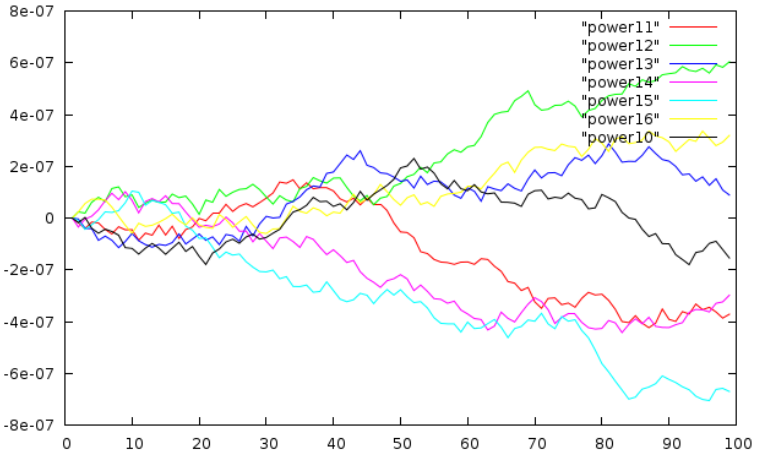
RAJ MALABAR
71 CASTLE STREET
CAMBRIDGE CB3 0AH
PHONE 01223312569
www.rajmalabar.com

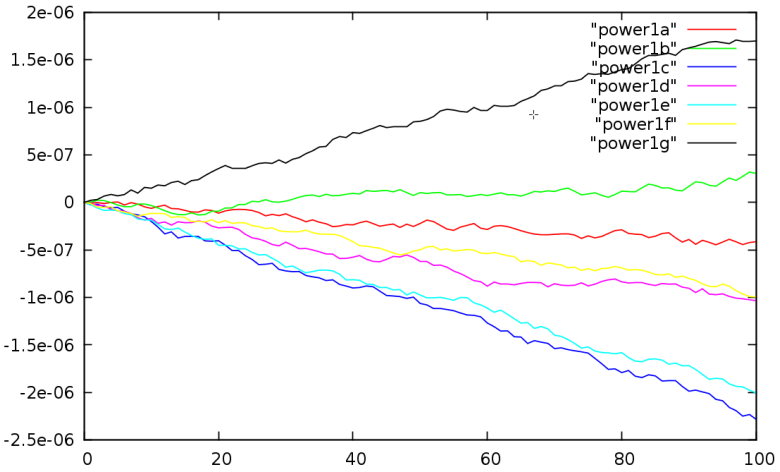
TICKET T001698 DATE 20/11/2007
WAITER 1 ROOM 1 TABLE 6

QTY	DESCRIPTION	PRICE	AMOUNT
1	King Fisher PT	2.75	2.75
1	King Fisher PT	2.75	2.75
2	Bitter PT	2.5	5
1	Seafood Biriyani	9.99	9.99
1	Chappathi	1.48999	1.48999
	Kerala Lamb Curry	8.28999	8.28999
	Porotta	2.49	2.49
	Coca Cola/ Diet Co	1.29	1.29
	Sweet/Salty Lassi	2.25	2.25
	erala Lamb Curry	8.28999	8.28999
	mon Rice	3.49	3.49
	ca Cola/ Diet Co	1.29	1.29
	cken Korma	7.99	7.99
	nut Rice	3.49	

Certain types of curry lead to problems afterwards.

Errors





Error amplification when some programs run for a long time ...

Errors

Patriot missile interceptor fails to intercept (1991) due to error amplification.

Numerical calculus

UNIVERSITY OF CAMBRIDGE

https://www.youtube.com/watch?v=EJI6GVQKHm8

Numerical integration and more error accumulation ...

UNIVERSITY OF CAMBRIDGE

THE $25,000,000,000* EIGENVECTOR

UNIVERSITY OF CAMBRIDGE

THE $25,000,000,000* EIGENVECTOR
THE LINEAR ALGEBRA BEHIND GOOGLE

UNIVERSITY OF CAMBRIDGE

THE $25,000,000,000* EIGENVECTOR
THE LINEAR ALGEBRA BEHIND GOOGLE

KURT BRYAN† AND TANYA LEISE‡

Abstract. Google's success derives in large part from its PageRank algorithm, which ranks the importance of webpages according to an eigenvector of a weighted link matrix. Analysis of the PageRank formula provides a wonderful applied topic for a linear algebra course. Instructors may assign this article as a project to more advanced students, or spend one or two lectures presenting the material with assigned homework from the exercises. This material also complements the discussion of Markov chains in matrix algebra. Maple and Mathematica files supporting this material can be found at www.rose-hulman.edu/∼bryan.

Key words. linear algebra, PageRank, eigenvector, stochastic matrix

Linear systems

UNIVERSITY OF CAMBRIDGE

THE $25,000,000,000* EIGENVECTOR
THE LINEAR ALGEBRA BEHIND GOOGLE

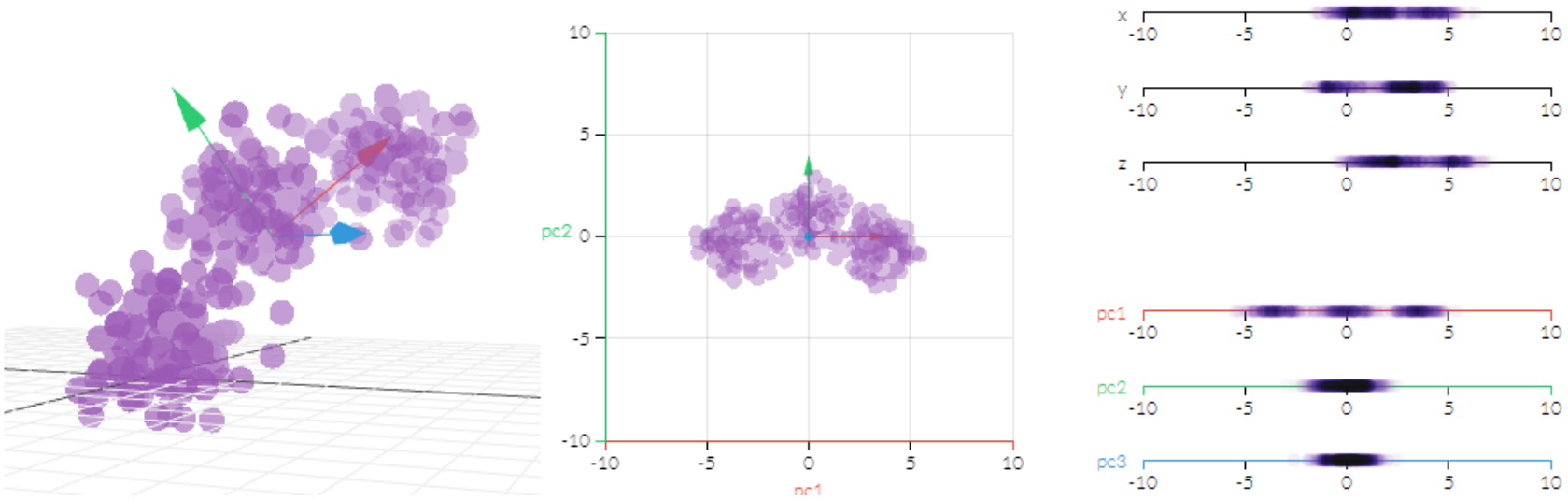
KURT BRYAN† AND TANYA LEISE‡

Abstract. Google's success derives in large part from its PageRank algorithm, which ranks the importance of webpages according to an eigenvector of a weighted link matrix. Analysis of the PageRank formula provides a wonderful applied topic for a linear algebra course. Instructors may assign this article as a project to more advanced students, or spend one or two lectures presenting the material with assigned homework from the exercises. This material also complements the discussion of Markov chains in matrix algebra. Maple and Mathematica files supporting this material can be found at www.rose-hulman.edu/∼bryan.

Key words. linear algebra, PageRank, eigenvector, stochastic matrix

* That was in 2004. It was $367b in 2015 (Forbes).

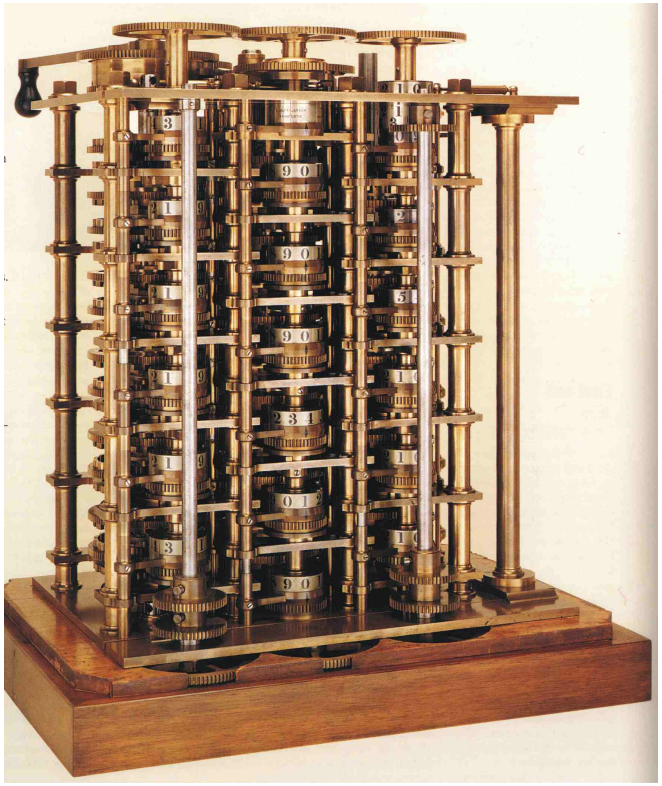
Data analysis



From *Explained Visually*

In a nutshell: How can we make sense of multi-dimensional data?

Number representation and floating point computation



Charles Babbage's machine used base 10. CompScis decided a little later that base 2 is more funky.

UNIVERSITY OF CAMBRIDGE

Intro to errors

Where do errors come from and why do we care?

UNIVERSITY OF CAMBRIDGE

Almost every computation we do will involve errors in some way:

- We may be working with imperfect input (e.g. noisy data)
- We may not be able to represent our values exactly
- Some of our algorithms knowingly produce the wrong answer so they can run faster

This is important because these errors can interact in non-obvious ways. If we are not careful, the errors can grow and leave us with nonsense ...

Absolute and relative errors

UNIVERSITY OF CAMBRIDGE

Intro to fundamental concepts. More on errors later in the course.

Let x be a real number. We will use x^* to denote its approximation. We define two ways of measuring error introduced by this approximation.

Absolute error: $\epsilon_x \equiv \Delta_x \equiv |x^* - x|$

When it is clear from the context we will use ϵ to denote the absolute error.

Relative error: $\eta_x \equiv \delta_x \equiv \frac{\Delta_x}{|x|} = \frac{|x^* - x|}{|x|}$

When it is clear from the context we will use η to denote the relative error.

The relationships (WARNING: Severe abuse of notation!)

$$x^* = x \pm \epsilon_x \quad \text{and} \quad x^* = x(1 \pm \eta_x).$$

are also commonly used to mean that x^* may take any value in the interval $[x - \epsilon_x, x + \epsilon_x]$. Note the abuse of the "$\pm$" notation.

Error Accumulation - Addition/Subtraction

UNIVERSITY OF CAMBRIDGE

Let

$$x^* = x \pm \epsilon_x \quad \text{and} \quad y^* = y \pm \epsilon_y.$$

Adding these two yields

$$\begin{aligned} x^* + y^* &= (x \pm \epsilon_x) + (y \pm \epsilon_y) \\ &= x + y \pm \epsilon_x \pm \epsilon_y \\ &= x + y \pm (\epsilon_x + \epsilon_y). \\ \epsilon_{x+y} &= \epsilon_x + \epsilon_y. \end{aligned}$$

Exercise: What about subtraction?

Error Accumulation - Addition/Subtraction

UNIVERSITY OF CAMBRIDGE

Beware: when addition or subtraction causes partial or total cancellation, the relative error of the result can be much larger than that of the operands. We call this **loss of significance**.

For example, consider we store values to 3 significant digits and we take the innocent-looking $x = 9.99, y = 9.98$. 3 significant figures means x and y are accurate to ± 0.005 absolute error. x and y thus each have a relative error of about 0.0005 (0.05%), i.e. very good.

However, $x - y = 0.01$, and has an absolute error of 0.01 (recall previous slide on subtraction) hence a relative error of 100%! We have little idea what the true value of $x - y$ is at this point.

Error Accumulation - Addition/Subtraction

UNIVERSITY OF CAMBRIDGE

This gets even worse when the loss of significance happens in a fraction's denominator. Consider an extension of the previous example:

$$\frac{1}{x-y} = \frac{1}{0.01 \pm 0.01}$$

This can be anywhere between 50 and infinity!

Unfortunately, lots of the problems we want to solve have this property. For example, inverting a matrix:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} = \frac{1}{ad-bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

There are many different combinations of values which can lead to a small $ad - bc$.

Error Accumulation - Addition/Subtraction

The previous examples were extreme: all of the significance was lost in one operation. This also made it relatively easy to spot the problem.

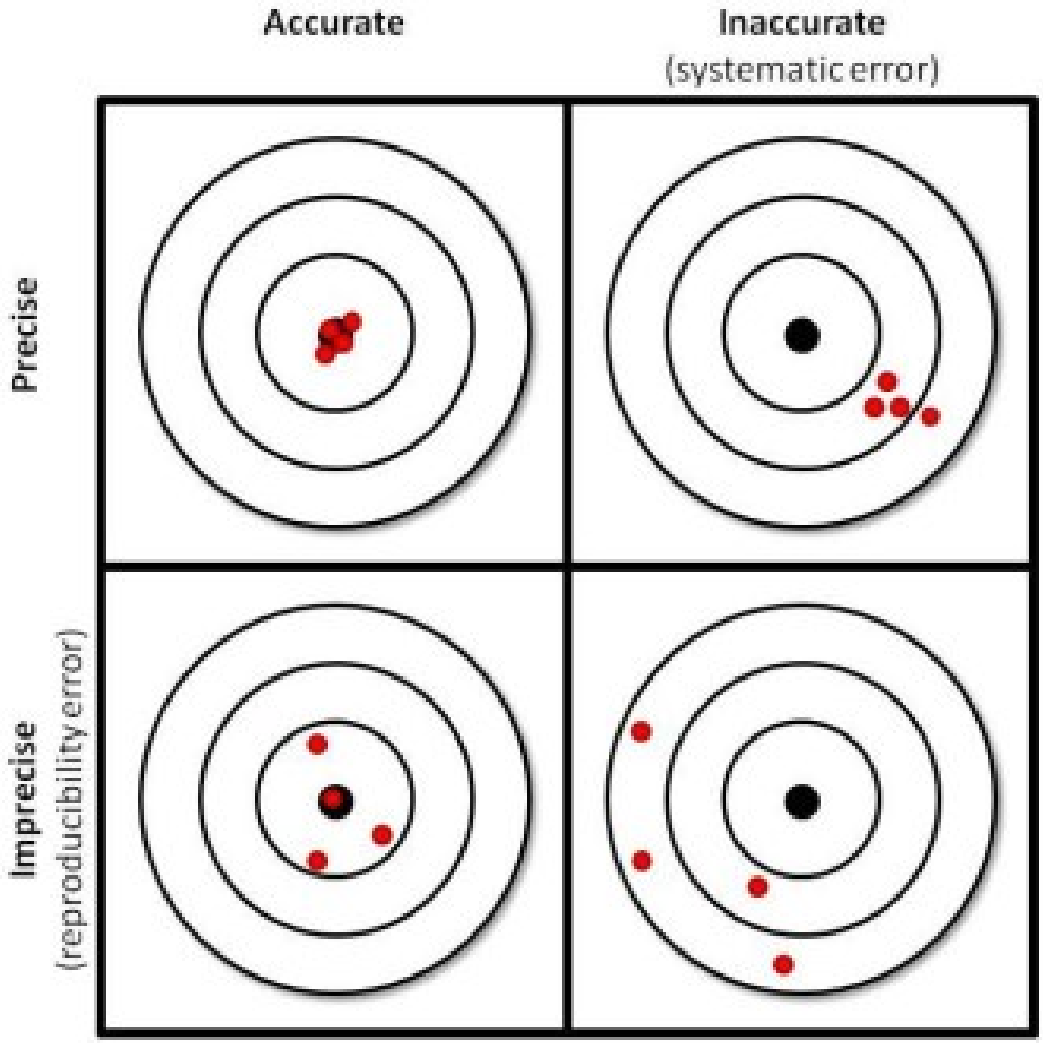
A far more insidious issue is **gradual loss of significance**. Consider this code:

```
double x_magic = 10.0/9.0; // x = 1.111111111 ...
double x = x_magic;
for (i=0; i<30; i++) {
    printf("%e\n", x);
    x = (x - 1.0) * 10.0;
}
```

Say initially x has 10 significant figures (sf) of accuracy. In every iteration in that loop it still stores 10sf, but the accuracy of the stored value reduces by 1sf per iteration.

It can be *very* hard to identify cases like this in the wild.

Accuracy vs Precision



Error Accumulation - Multiplication/Division

UNIVERSITY OF CAMBRIDGE

Let

$$x^* = x(1 \pm \eta_x) \quad \text{and} \quad y^* = y(1 \pm \eta_y).$$

What is the relative error of xy? To find η_{xy} we aim to obtain an equation of the form $x^*y^* = xy(1 \pm \eta_{xy})$ and then identify terms.

$$\begin{aligned} x^*y^* &= x(1 \pm \eta_x) \times y(1 \pm \eta_y) \\ &= xy(1 \pm \eta_x \pm \eta_y \pm \eta_x\eta_y) \\ &\approx xy(1 \pm \eta_x \pm \eta_y) \\ &= xy(1 \pm (\eta_x + \eta_y)). \\ \eta_{xy} &\approx \eta_x + \eta_y, \end{aligned}$$

where we assumed η_x and η_y are small enough such that $\eta_x\eta_y$ is negligible.

Exercise: What about division?

Error Propagation

UNIVERSITY OF CAMBRIDGE

Previously we explored what happens with the error when we apply basic arithmetic operations.

If we have a function relationship, i.e. we wish to evaluate $f^*(x)$ as an approximation for $f(x)$ when x is approximated by x^* then we use the first derivative. The absolute error is:

$$\Delta_{f(x)} \approx |f'(x^*)|\Delta_x,$$

Notice anything interesting (or worrying)? Let's look at an example.

Error Propagation

UNIVERSITY OF CAMBRIDGE

For example, let

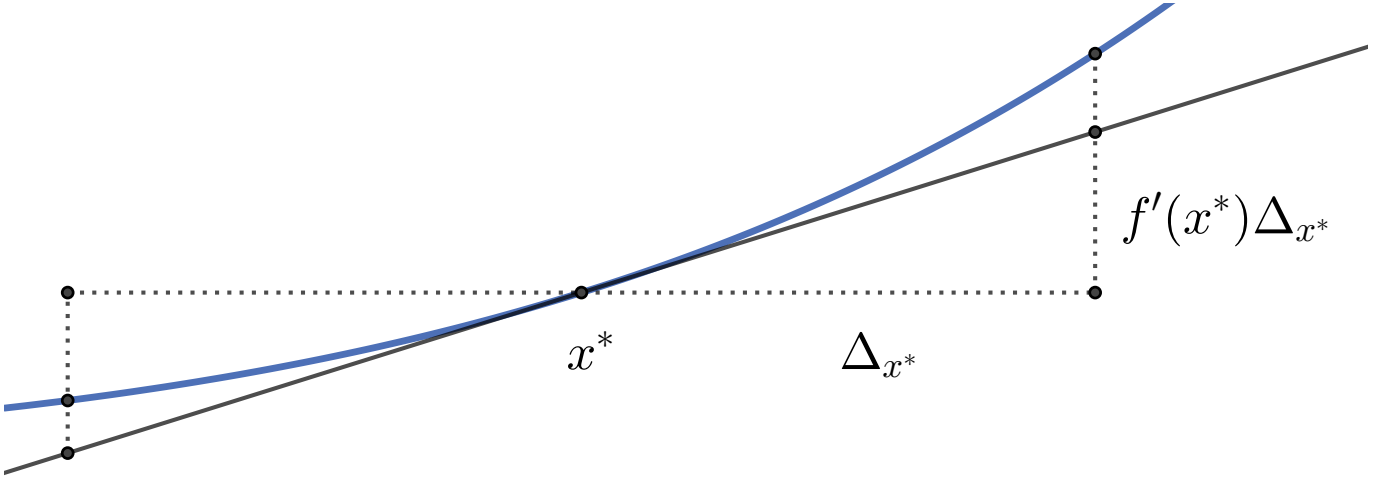
$$f(x) = x^2.$$

We then have:

$$\Delta_{f(x)} \approx |f'(x^*)|\Delta_x = |2x^*|\Delta_x.$$

The error depends not only on the error at the input but also on the value of the input! Higher values of x will amplify the absolute error.

Error Propagation Visually



$$\Delta_{f(x)} \quad = \quad \max_{x^*-\Delta_x \leqslant t \leqslant x^*+\Delta_x} |f(x) - f(t)| \quad \approx \quad |f'(x^*)|\Delta_x$$

The smaller Δ_{x^*} the more accurate the approximation (recall how dx and dy are defined?)

Error Bounds

The above smells of Taylor. We can get tighter error bounds by using the Taylor theorem. Assuming the first $n+1$ derivatives of f are continuous and bounded in the vicinity of x^*, then the error of truncating the infinite Taylor series about x^* to the first n terms is bounded by the Lagrange error bound:

$$\Delta_{f(x)} = |f(x) - f^*(x)| \quad \leq \quad M \cdot \frac{|x - x^*|^{n+1}}{(n+1)!} = M \cdot \frac{\Delta_x^{n+1}}{(n+1)!},$$

where $f^*(x)$ is our truncated Taylor series and M is an upper bound (ideally the smallest upper bound) on the $(n+1)$th derivative of f on the interval we wish the bound the error, i.e. $M \geq \max_{\zeta \in (\min(x^*,x), \max(x^*,x))} |f^{(n+1)}(\zeta)|$.

We shall derive the above important expression later in the course.

Exercise: Find the error bound for estimating $e^{0.3}$ with a 3rd degree Taylor polynomial for $f(x) = e^x$ about $x^* = 0$.

Error Bounds in Practice

UNIVERSITY OF CAMBRIDGE

Importantly, we seldom know the *exact* error in a measurement or a program, otherwise we could calculate the floating point answer and correct it get a mathematically perfect answer.

Instead, we can analyse the problem and extract *an error bound* but, unlike the previous example, sometimes it's impossible to do so analytically.

Some problems are incredibly complex for which the only way to extract error bound information is to resort to **Monte Carlo** techniques: try sufficiently numerous input values (either randomly, or from those of interest) and sufficiently numerous random perturbations for each, and extract the error bounds as the maximum difference between output values at every try; as seen before, these may depend on the input value, and you'd obtain an error dependency profile.

Error Analysis

UNIVERSITY OF CAMBRIDGE

Forward error analysis examines how perturbations of the input propagate.

For example, in the case of the function $f(x) = x^2$ the relative error approximately doubles. The previous examples are examples of forward error analysis.

Forward error analysis sometimes leads to pessimistic overestimates of the error, especially when a sequence of calculations is considered and in each calculation the error of the worst case is assumed (perfectly sensible!).

In practice, errors *sometimes* average out (i.e. an error in one calculation gets reduced by a later error of opposite sign) but don't just assume that.

It's best to analyse and determine the error bounds of your problem in order to determine if improvements are necessary. You might be lucky (not!).

Error Analysis

UNIVERSITY OF CAMBRIDGE

Backward error analysis examines the question *"how much error in input would be required to describe a given error in the output?"* It assumes that an approximate solution to a problem is good if it is the exact solution to a nearby problem. For the previous example, the output can be written as

$$f^*(x) = \left(x^2\right)^* = x^2(1+\eta)$$

where η denotes the relative error in the output. Assuming η is sufficiently small, we have $1+\eta > 0$. Thus there exists $\tilde{\eta} < \eta$ such that

$$(1+\tilde{\eta})^2 = 1+\eta,$$

and

$$f^*(x) = x^2(1+\tilde{\eta})^2 = f(x(1+\tilde{\eta})),$$

i.e. if the backward error is small, we accept the solution, since it is the correct solution to a nearby problem.

Revision: O-notation

There are multiple situations in this course where we are interested in bounding a function f by a (hopefully simpler) function g. This makes it easier to compare and classify behaviour.

- **Execution time of algorithms**, with $f(x) =$ execution time for an input of size x.
- **Convergence towards a solution**, with $f(\epsilon) =$ new error in terms of old error.

We say that $f(x) = O(g(x))$ if there exists a real constant $M > 0$ and an x_0 such that

$$|f(x)| \leq Mg(x) \text{ for all } x \geq x_0.$$

Although $g(x)$ can take various forms, in this course you'll mostly meet simple polynomial functions, e.g. x^2, x^3, etc.

Numerical Differentiation

Introduction

UNIVERSITY OF CAMBRIDGE

Numerical differentiation is the procedure of (numerically) approximating the value of a derivative of a given function at a given point using values of the function (and possibly other knowledge about the function).

Computing a derivative is one of the most common tasks faced in numerical computations. Some of them include:

- iterative methods (e.g. Newton-Raphson, introduced later in this course);
- solving differential equations;
- computational geometry;
- computer graphics, etc.

Taylor Series

Taylor is a great friend to have! Suppose f is infinitely differentiable in the vicinity of a point x_0 then its Taylor series/expansion about x_0 is given by

$$f(x) = f(x_0) + \frac{f'(x_0)}{1!}(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \dots \tag{1}$$

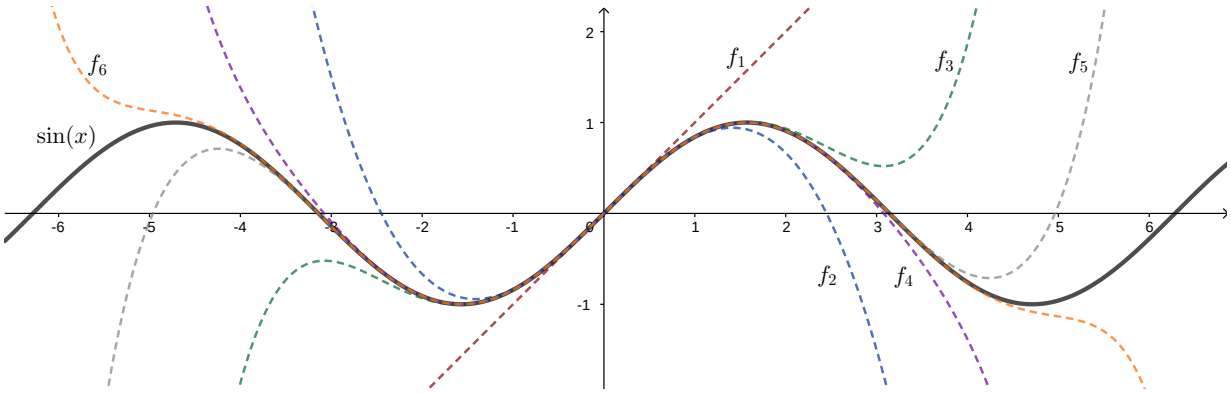
$$= \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x - a)^n \tag{2}$$

We're going to consider functions f whose Taylor series converges to f.

Note for pedantics: Taylor series and Taylor theorem are not one and the same. The latter gives an approximation of a k-times differentiable function and deals with the remainder.

(Note the pedantics #2: A convergent Taylor series of f does not necessarily converge to f! Look up non-analytic functions.)

The classic example: $\sin x$ about $x_0 = 0$:



The more terms we consider, the more accurate the approximation. More on truncation errors soon.

First derivative

UNIVERSITY OF CAMBRIDGE

In the first approximation, we neglect the terms starting from the second derivative, i.e. we truncate the series at the first two terms:

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0),$$

yielding an approximation of the first derivative.

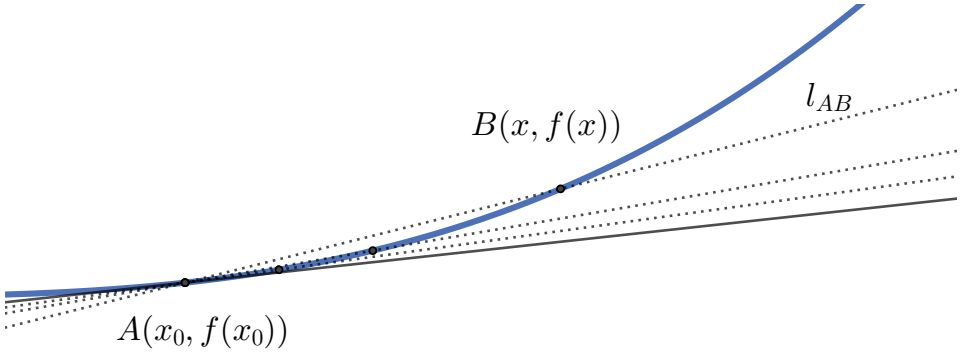
$$f'(x_0) \approx \frac{f(x) - f(x_0)}{x - x_0}.$$

Note: Obviously, don't confuse the above with the definition of the derivative at a point, despite its similar looks.

Next we'll see alternative ways to obtain approximations for the first derivative.

Geometric Representation

Recall the geometric interpretation of the first derivative. Given a differentiable function f, $f'(x_0)$ is the slope of the tangent to the curve described by f at the point x_0.



For a x sufficiently close to x_0 we can approximate that tangent with the line going through corresponding points A and B, i.e.

$$f'(x_0) \approx \text{slope}(l_{AB}) = \frac{y_A - y_B}{x_A - x_B} = \frac{f(x) - f(x_0)}{x - x_0}.$$

UNIVERSITY OF CAMBRIDGE

Or, simply, straight from the definition of the derivative,

$$f'(x_0) = \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0},$$

then, if x is sufficiently close to x_0, we have

$$f'(x_0) \approx \frac{f(x) - f(x_0)}{x - x_0}.$$

Note the notion of *"sufficiently close"* - what does it mean? Can we quantify/have control over it?

First Derivative

UNIVERSITY OF CAMBRIDGE

Let's denote the difference between x and some point y in its vicinity with $h = y - x$. Given a parameter h and a function f we can approximate its first derivative using the following relation

$$D^+_{f'}(x) \equiv \frac{f(x+h) - f(x)}{h}.$$

We will call h the *discretisation parameter*. Clearly, $D^+_{f'}(x)$ heavily depends on the discretisation parameter h. Next we'll see how to describe this dependency.

$D^+_{f'}$ is usually called the *forward* approximation.

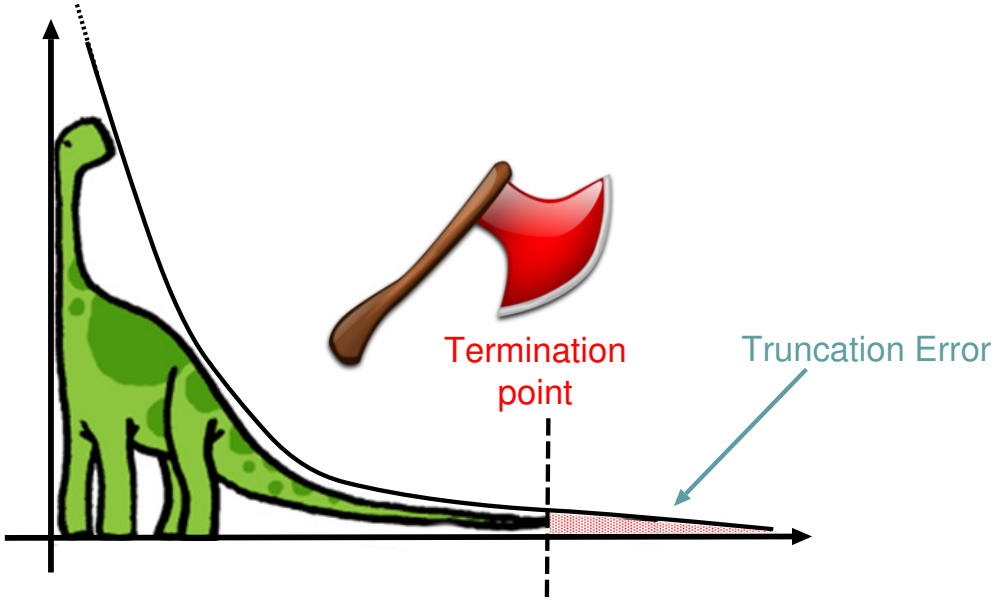
Rounding versus Truncation Error

UNIVERSITY OF CAMBRIDGE

Two distinct types of error in calculations are:

Rounding error the error we get by using finite arithmetic for computation (e.g. finite number of digits). More on this later on in the course.

Truncation error (a.k.a. discretisation error) the error we get by stopping an infinite process after a finite point.



Note the general antagonism: the finer the mathematical approximation the more operations needed, and hence the worse the accumulated error (potentially). Need to compromise, or use/design *very* clever algorithms (beyond this course).

Truncation Error - Example

UNIVERSITY OF CAMBRIDGE

Given a discretisation parameter h, we want to compute how much error we introduce by using the approximation $D^+_{f'}$ when computing a first derivative, i.e. truncation error.

The truncation error can be derived from Taylor's theorem:

$$\begin{aligned} f(x+h) &= f(x) + hf'(x) + h^2 f''(x)/2! + O(h^3) \\ f'(x) &= -f(x)/h + f(x+h)/h - hf''(x)/2 - O(h^2) \\ f'(x) &= D^+_{f'}(x) - hf''(x)/2 - O(h^2) \\ f'(x) - D^+_{f'}(x) &= -hf''(x)/2 + O(h^2) \end{aligned}$$

For a sufficiently small h, $|O(h^2)|$ is negligible comparing to $|-hf''(x)/2|$, hence we can ignore it.

The term

$$-hf''(x)/2$$

is linear in h – halving h will halve the truncation error (approximately, since we ignored the Taylor remainder).

This, on its own, may not be a very useful indicator, but it is very useful when comparing two methods.

Another Example

UNIVERSITY OF CAMBRIDGE

Here's an alternative form of numerical differentiation:

$$D^0_{f'} \equiv \frac{f(x+h) - f(x-h)}{2h}.$$

This form is usually called the *central* or *symmetric* approximation. There is also a $D^-_{f'}(x) \equiv (f(x) - f(x-h))/h$ called the *backward* approximation.

How can we compare $D^+_{f'}$ to $D^0_{f'}$?

Another Example (cont'd)

UNIVERSITY OF CAMBRIDGE

Taylor to the rescue:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{3!}f'''(x) + O(h^4)$$
$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{3!}f'''(x) + O(h^4)$$

Subtracting gives

$$f(x+h) - f(x-h) = 2hf'(x) + 2h^3 f'''(x) + O(h^4).$$

Now, the truncation error is

$$\frac{f(x+h) - f(x-h)}{2h} - f'(x) = \frac{h^2 f'''(x)}{3!} + O(h^3).$$

Another Example (cont'd)

UNIVERSITY OF CAMBRIDGE

So, the truncation of error of $D^0_{f'}$ is approximately

$$h^2 f'''(x)/3!,$$

which is quadratic in h – halving h gives a $1/4$x truncation error (approximately, since we again ignored the Taylor remainder).

Hence, $D^0_{f'}$ is a much better approximation than $D^+_{f'}$ in terms of truncation error.

$D^+_{f'}$ vs. $D^0_{f'}$

These examples for approximating the first derivative illustrate an important concept regarding approximations in general.

Formula	$\frac{f(x+h)-f(x)}{h}$	$\frac{f(x+h)-f(x-h)}{2h}$
Error	$\frac{hf''(x)}{2}+O(h^2)$	$\frac{h^2f'''(x)}{3!}+O(h^3)$
Order	First	Second

Often there are multiple algorithms which converge to the same theoretical limit (such as the two differentiation examples above), but which have different *rates of approaching that limit*.

(They may also have different rounding error accumulation but we're ignoring that for now.)

Order of Approximation

UNIVERSITY OF CAMBRIDGE

An approximation method with parameter $h \to 0$ has order n if the truncation error in the approximation is proportional to $O(h^n)$. This is sometimes called an nth order method.

For example, $D^+_{f'}$ is a first-order method of approximating derivatives of well-behaved functions and $D^0_{f'}$ is a second-order method.

Much effort during past decades was invested in devising higher-order methods so that larger h can be used without incurring excessive truncation error. This can also reduce the impact of rounding errors, since we can stop earlier before rounding errors become important (relatively speaking, as quantities decrease).

NOTE: *"Order of approximation"* is not the same as *"order of convergence"* (which we'll see later on in the course). The former quantifies the approximation error. The latter quantifies convergence speed.

Numerical Integration

Numerical integration

UNIVERSITY OF CAMBRIDGE

We have already seen how we might approximate a function's derivative. Now let's look at approximating its integral between two limits.

We might need to do this because:

- We only know the value of $f(x)$ at a subset of points.
- It is difficult or impossible to compute the antiderivative (primitive) function analytically.
- We have a function for the integral, but it is expensive to compute (e.g. an infinite series), so it is cheaper to compute an approximation.

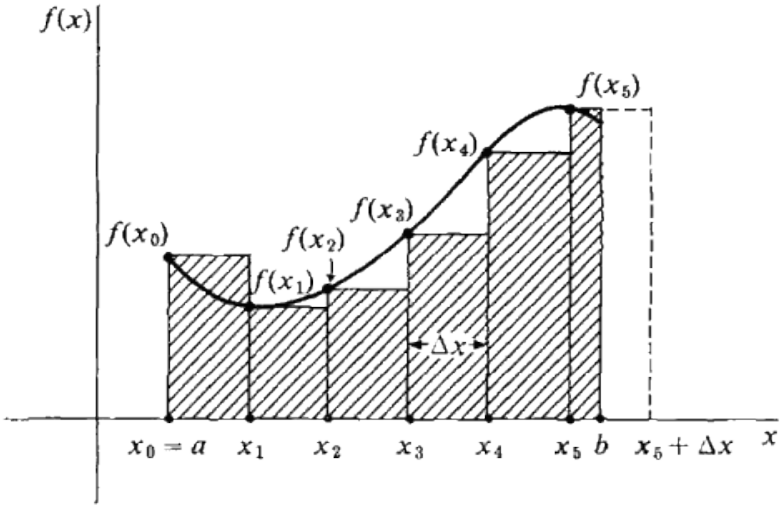
Methods for numerical integration are often called **quadrature techniques** (historical, from computing areas in geometry by "squaring things").

Riemann integral

The first rigorous definition of an integral between two limits was the **Riemann integral**.

$$\int_a^b f(x)dx$$

The interval $[a, b]$ is split into sub-intervals. The area within each sub-interval is computed by assuming the function to be constant on the sub-interval. By adding together all of these small areas, we get a **Riemann sum**.

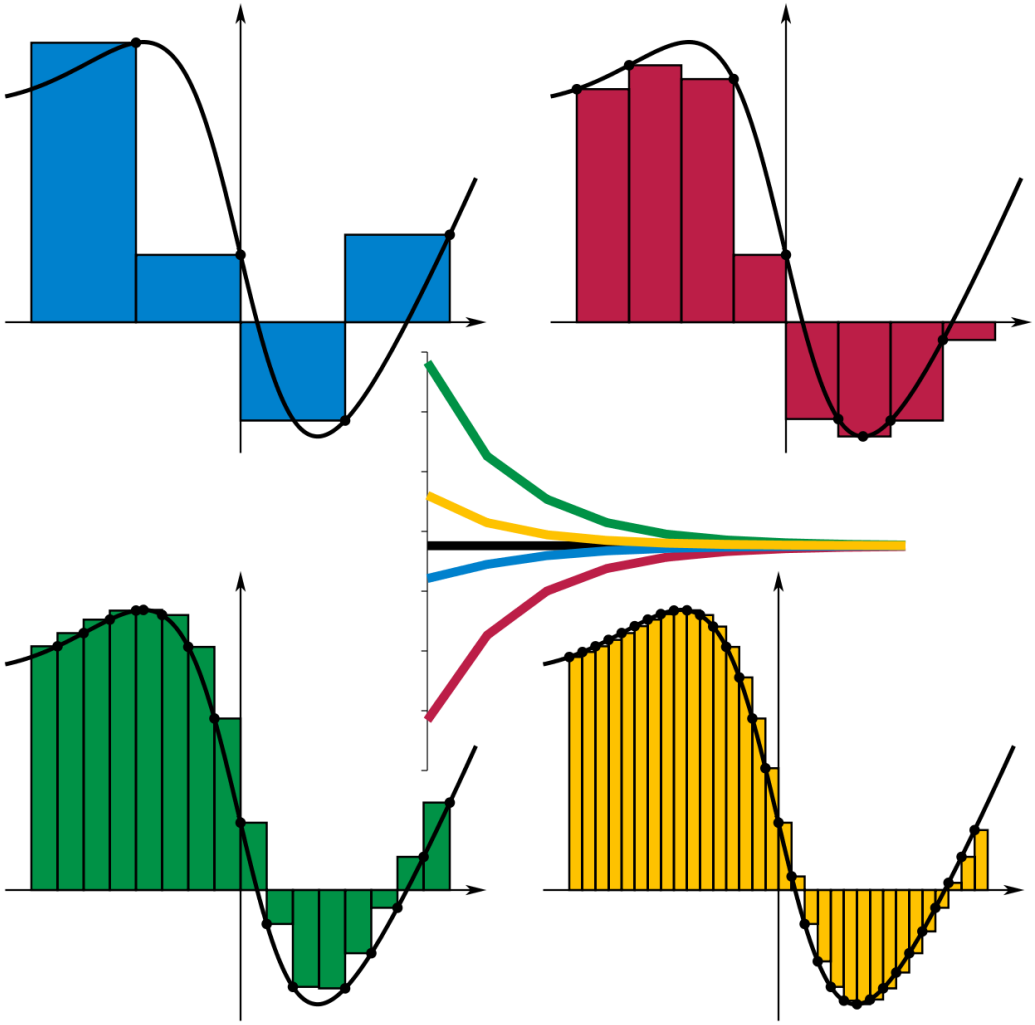


$$a = x_0 < x_1 < ... < x_n = b$$

$$t_i \in [x_i, x_{i+1}] \text{ for } i \in [0, n-1]$$

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} f(t_i)(x_{i+1} - x_i)$$

Riemann integral



From Wikipedia

`https://en.wikipedia.org/wiki/Riemann_sum`

Riemann integral

UNIVERSITY OF CAMBRIDGE

As the size of each sub-interval approaches zero, the Riemann sum approaches the Riemann integral. A possible general definition is:

$$\int_a^b f(x)dx = \lim_{n\to\infty} \sum_{i=0}^{n} f(x_i)\ \frac{b-a}{n},$$

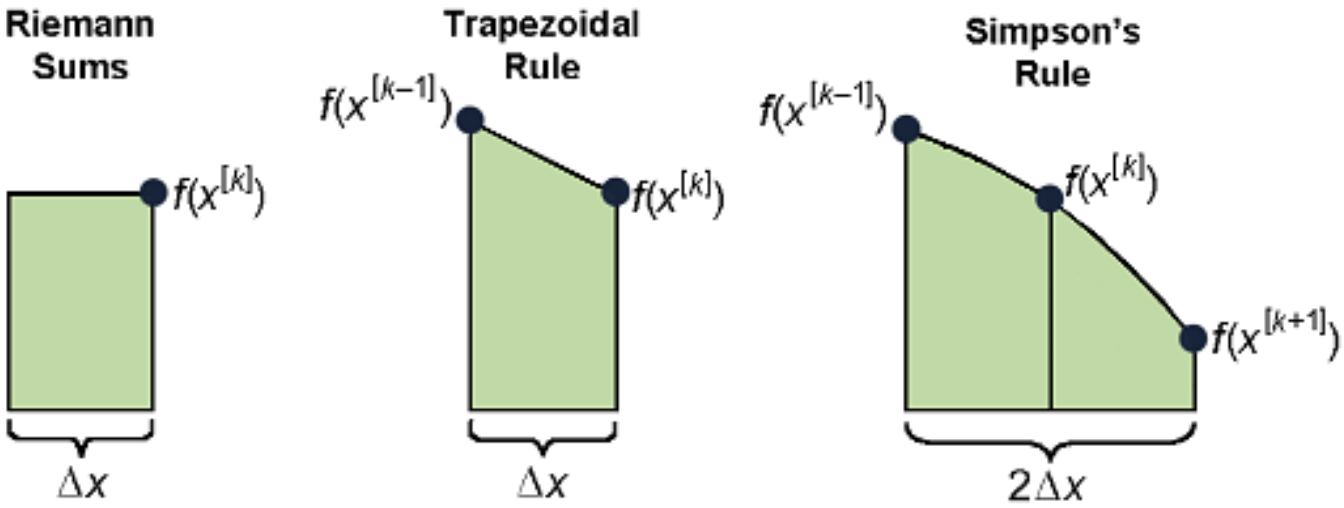
where $x_i = a + (b-a)i/n$. This uses equally sized sub-intervals.

Obviously, as the size of each sub-interval approaches zero, the number of sub-intervals tends to ∞, so this is not suitable for numerical computations. Numerically, we try to approximate it by stopping at a finite interval size.

Note that there is no requirement for the sub-intervals to be sized equally, or for the representative points to be in particular locations within their sub-intervals (start, midpoint, end, etc.). In the limit, all of these factors become negligible.

Commonly-used quadrature techniques

Generally we fit some spline to the data and find the area under the spline.



Mid-point rule - horizontal line passing through the midpoint of each pair of coordinates to make rectangular strips. There are also the **Left-point** rule and **Right-point rule** (pictured). These are the middle, left and right Riemann sums.

Trapezium rule - secant through each pair of adjacent values.

Simpson's rule - parabola segment through each three consecutive values.

Commonly-used quadrature techniques

UNIVERSITY OF CAMBRIDGE

If we split the interval $[a, b]$ into n equal subintervals of length $h = \dfrac{b-a}{n}$, then we have $n+1$ equally spaced coordinates $x_i = a + ih$, where $i = 0, 1, \ldots, n$.

- The mid-point rule treats $f(x)$ as piecewise **constant** (0th order polynomial).

$$\int_a^b f(x)dx = h \sum_{i=0}^{n-1} f\left(\frac{x_i + x_{i+1}}{2}\right).$$

- The trapezium rule treats $f(x)$ as piecewise **linear** (1st order polynomial).

$$\int_a^b f(x)dx = h \sum_{i=0}^{n-1} \frac{f(x_{i+1}) + f(x_i)}{2}.$$

- Simpson's rule treats $f(x)$ as piecewise **quadratic** (2nd order polynomial):

$$\int_a^b f(x)\,dx \overset{\text{Simpson}}{\approx} \frac{h}{3}\left(f(x_0) + 2\sum_{j=1}^{n/2-1} f(x_{2j}) + 4\sum_{j=1}^{n/2} f(x_{2j-1}) + f(x_n)\right),$$

where n here must be **even**. Rounding error is a random walk proportional to $\sqrt{n}$. Truncation error depends on $f(x)$: it is zero for quadratics and several classes of higher-order polynomial.

Simpson's rule is a fourth-order method, so we might expect best results with $h \approx (b-a)\sqrt[5]{\epsilon_0}$. Here ϵ_0 is the machine epsilon, discussed later on in the course.

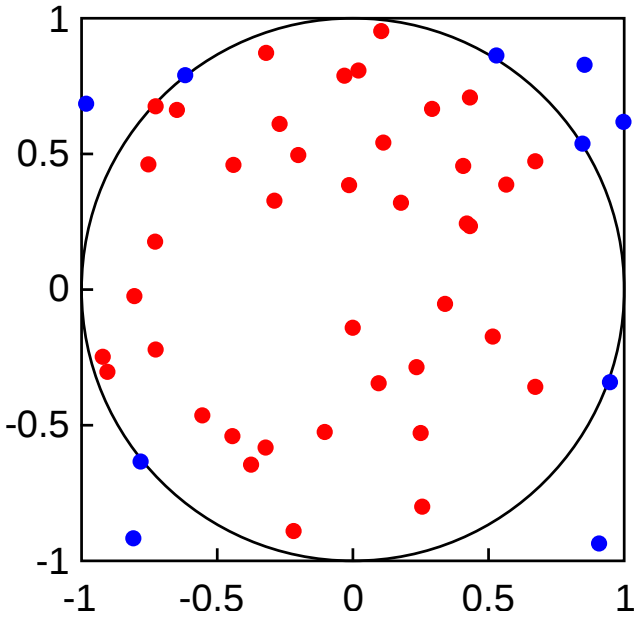
Warning: Note: Don't be tempted to assume that these are similar to truncating more terms in a Taylor series, i.e. that higher order polynomial interpolation necessarily gives a better approximation of the integral! Those can oscillate. See *Runge's phenomenon*, `https://en.wikipedia.org/wiki/Runge%27s_phenomenon`.

Monte Carlo integration

Monte Carlo methods are algorithms that use **random sampling** to produce a result. We can use this for integration, by randomly sampling a known domain D, rejecting samples that do not satisfy a set of analytic conditions corresponding to the area we want to integrate.

Example: Estimate the area of a circle with centre and radius (x_0, y_0) and R.

1. Create a **domain** D (the bounding box) whose area is easy to compute.
2. Randomly sample $(x_i, y_i) \in D$, recording whether (x_i, y_i) satisfies $(x_i - x_0)^2 + (y_i - y_0)^2 \leq R^2$.
3. $\text{Area}_C = \text{Area}_D \times \dfrac{\text{num\_points}_C}{\text{num\_points}_D}$



This method is particularly useful for high-dimensional integrals.

Iterative Methods

UNIVERSITY OF CAMBRIDGE

Why iterate?

We often do not have the means to directly compute the solution to a problem, so it may be easier to start with a guess and iteratively refine it. For example:

- When direct methods are too expensive.
  (e.g. Testing every possibility in a large search space to find the best)
- When we don't know how to solve the problem analytically.
  (e.g. Finding roots of high-degree polynomials)
- When we don't know how to formulate the problem mathematically.
  (e.g. Finding cats in images)

Introduction to iterative methods

UNIVERSITY OF CAMBRIDGE

All iterative methods need two main components:

Iterative step

A way of deriving a new guess from the previous one(s)

Termination criteria

A way of deciding when to stop iterating

There are a few obvious choices for termination criteria. They can be combined to suit the application.

- **Absolute error** drops below some threshold
- **Relative error** drops below some threshold
- **Some "measure" between successive iterations** is below a threshold
- **Iteration count** exceeds some threshold
- **Computation time** exceeds some threshold

(This is not an exhaustive list.)

UNIVERSITY OF CAMBRIDGE

How do we measure error when we don't know the goal?

There are a couple of options:

- **Substitute our guess back into the equation.**
  e.g. If the unknown root of f is r, instead of measuring the error between our guess x and r, we can measure the error between $f(x)$ and $f(r) = 0$.
- **Test the latest estimate** x_n, measuring the error between it and the previous guess x_{n-1}.
  This is not 100% reliable because there are some methods which can settle on the same value for multiple iterations before continuing to converge, but it is commonly used.

Introduction to iterative methods

UNIVERSITY OF CAMBRIDGE

Despite being useful in a wide range of situations, there are some drawbacks to iterative methods:

- There are many situations where convergence is slow, and these situations can be difficult to predict or avoid.
- We are not guaranteed to converge to the solution we wanted. In some cases we are not guaranteed to converge at all!
- Errors can accumulate – particularly when we are not able to represent our numbers exactly.

Caution is required.

Order of convergence

UNIVERSITY OF CAMBRIDGE

For methods that aim to converge to a solution it is useful to quantify **how quickly** they do so. In other words, we'd like to express the error of one iteration in terms of the error of the previous iteration:

$$\epsilon_n = f(\epsilon_{n-1})$$

where $\epsilon_n = |r - x_n|$ is the error between the nth estimate x_n and the solution r.

kth order convergence means that $\epsilon_n = M\epsilon_{n-1}^k$ when $n \to \infty$ for some finite constant $M > 0$. Alternatively, we can write $\lim_{n\to\infty} \frac{\epsilon_n}{\epsilon_{n-1}^k} = M$.

Often it is convenient to investigate if $\epsilon_n = M\epsilon_{n-1}^k + O(\epsilon_{n-1}^{k+1})$, which is a sufficient condition for kth order convergence, and write $\epsilon_n = O(\epsilon_{n-1}^k)$.

We write simply $\epsilon_n \approx M\epsilon_{n-1}^k$ to mean kth order convergence.

Order of convergence

UNIVERSITY OF CAMBRIDGE

- **First-order** (linear) **convergence:** $\epsilon_n \approx M\epsilon_{n-1}$ or sufficiently
  $\epsilon_n = M\epsilon_{n-1} + O(\epsilon_{n-1}^2)$
  (linear dependency between errors of successive iterations; slow in practice)
- **Second-order** (quadratic) **convergence:** $\epsilon_n \approx M\epsilon_{n-1}^2$ or sufficiently
  $\epsilon_n = M\epsilon_{n-1}^2 + O(\epsilon_{n-1}^3)$
  (quadratic dependency between errors of successive iterations)
  i.e. The number of final significant digits approximately doubles with every iteration (much faster than first-order convergence).
- **Superlinear convergence:** $\epsilon_n \approx M\epsilon_{n-1}^k$ where $1 < k < 2$ (not as fast as quadratic, but acceptable in practice)
- etc.

We shall look at some examples in the next few slides.

Root finding

UNIVERSITY OF CAMBRIDGE

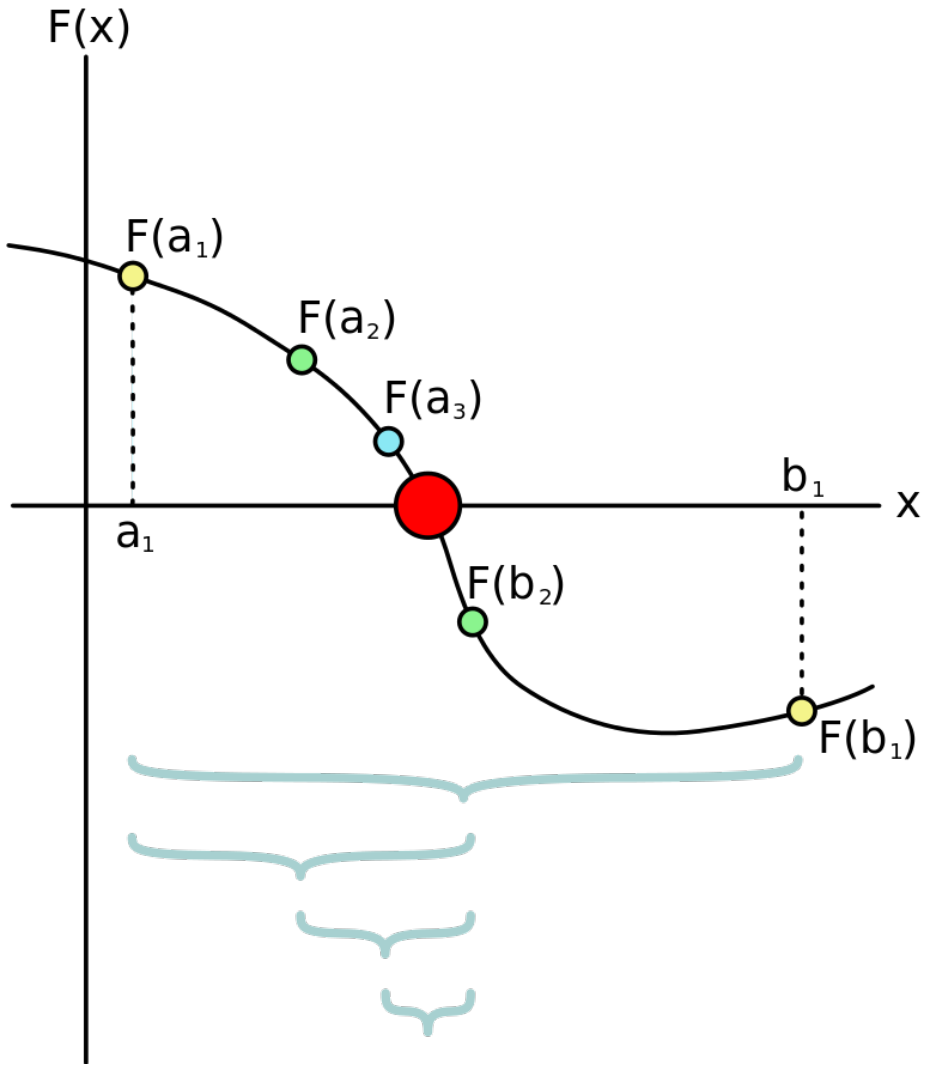
Commonly we want to find x such that $f(x) = 0$, i.e. a root of f. This is an enormously important problem. Many real-world problems are intrinsic root finding problems (including many minimisation problems, curve intersection etc.).

We'll cover three ways of doing this:

1. Bisection method
2. Fixed-point iteration
3. Newton-Raphson method (a special case of fixed-point iteration)

For example, we know that the "golden ratio" $\phi = \dfrac{1+\sqrt{5}}{2} \approx 1.618$ satisfies $\phi^2 = \phi + 1$, and we can compute it numerically as the positive root of $f(x) = x^2 - x - 1$.

Bisection method



Bisection method

UNIVERSITY OF CAMBRIDGE

The Bisection Method, a form of **successive approximation**, is a simple and robust approach.

1. Choose initial values a, b such that $\text{sign}(f(a)) \neq \text{sign}(f(b))$
   (Is this trivial for a computer?)
2. Find mid point $c = \dfrac{a+b}{2}$
3. If $|f(c)| <$ `desired_accuracy` then stop
4. If $\text{sign}(f(c)) = \text{sign}(f(a))$ then $a = c$ else $b = c$
5. Goto 2

Bisection method

UNIVERSITY OF CAMBRIDGE

To find the golden ratio, let $f(x) = x^2 - x - 1$.

i	a	b	c	$\text{sign}(f(c))$	err	err/prev. err
0	1.0	2.0	1.5	-1	1.18E-01	
1	1.5	2.0	1.75	1	1.32E-01	1.12E+00
2	1.5	1.75	1.625	1	6.97E-03	5.28E-02
3	1.5	1.625	1.5625	-1	5.55E-02	7.97E+00
4	1.5625	1.625	1.59375	-1	2.43E-02	4.37E-01
5	1.59375	1.625	1.609375	-1	8.66E-03	3.57E-01
6	1.609375	1.625	1.6171875	-1	8.46E-04	9.78E-02
7	1.6171875	1.625	1.62109375	1	3.06E-03	3.61E+00
8	1.6171875	1.62109375	1.619140625	1	1.11E-03	3.62E-01
9	1.6171875	1.619140625	1.6181640625	1	1.30E-04	1.18E-01
10	1.6171875	1.6181640625	1.61767578125	-1	3.58E-04	2.75E+00

Bisection method

UNIVERSITY OF CAMBRIDGE

The absolute error is halved at each step (i.e. $\epsilon_{n+1} \approx \frac{1}{2}\epsilon_n$) so the bisection method has **first-order convergence**. It is also known as binary chop (gives one bit per iteration, as we'll see later on in the course).

First-order convergence requires a number of steps proportional to the number of digits wanted, i.e. to the logarithm of the desired numerical precision.

Exercise: Compute the minimum number of iterations required by the bisection method to converge to the root with maximum absolute error ϵ.

Exercise: When can the bisection method encounter problems?

Fixed-point iteration

UNIVERSITY OF CAMBRIDGE

We wish to find a root of $f(x)$, i.e. solve $f(x) = 0$. This can be rewritten as $g(x) = x$, for some appropriate g, such that any solution to this second equation is a solution to the original equation. We call such a solution *a fixed point of* g, and it can be found numerically under certain conditions. The theory of fixed-point iteration gives us theoretical tools to better analyse convergence of algorithms.

Algorithm: Set $x = g(x)$ and generate the sequence $(x_n) = x_0, x_1, x_2, \ldots$ such that $x_{n+1} = g(x_n)$. If the sequence (x_n) converges to some r, i.e. $\lim_{n\to\infty}(x_n) = r$, then r is a fixed point of g and hence also a root of f.

Exercise: Use the above algorithm to approximate a root of $x^3 - 7x + 2 = 0$ for $x \in [0, 1]$. Hint: Take $g(x) = (x^3 + 2)/7$.

Fixed-point iteration

UNIVERSITY OF CAMBRIDGE

Convergence. Does the previous algorithm always converge? No. It depends on the choice of both $g(x)$ and of x_0. Example: Try the previous exercise for $x_0 = 2.5$. So when are we guaranteed to converge?

Convergence criterion: If $g : I \to I$ (maps I onto itself) and is differentiable on I such that $|g'(x)| < 1$ for all $x \in I$ then g has exactly one fixed point r in I and the sequence (x_n) defined previously with any $x_0 \in I$ will converge to r.

The proof is left as an exercise (*Hint:* The mean value theorem is handy) but realise first that if $g : I \to I$ then for any $x_0 \in I$ we have $g(x_0) = x_1 \in I$ and $g(g(x_0)) = x_2 \in I$ and so on. In other words, the entire sequence (x_n) is guaranteed to be contained by I if $x_0 \in I$.

Fixed-point iteration

UNIVERSITY OF CAMBRIDGE

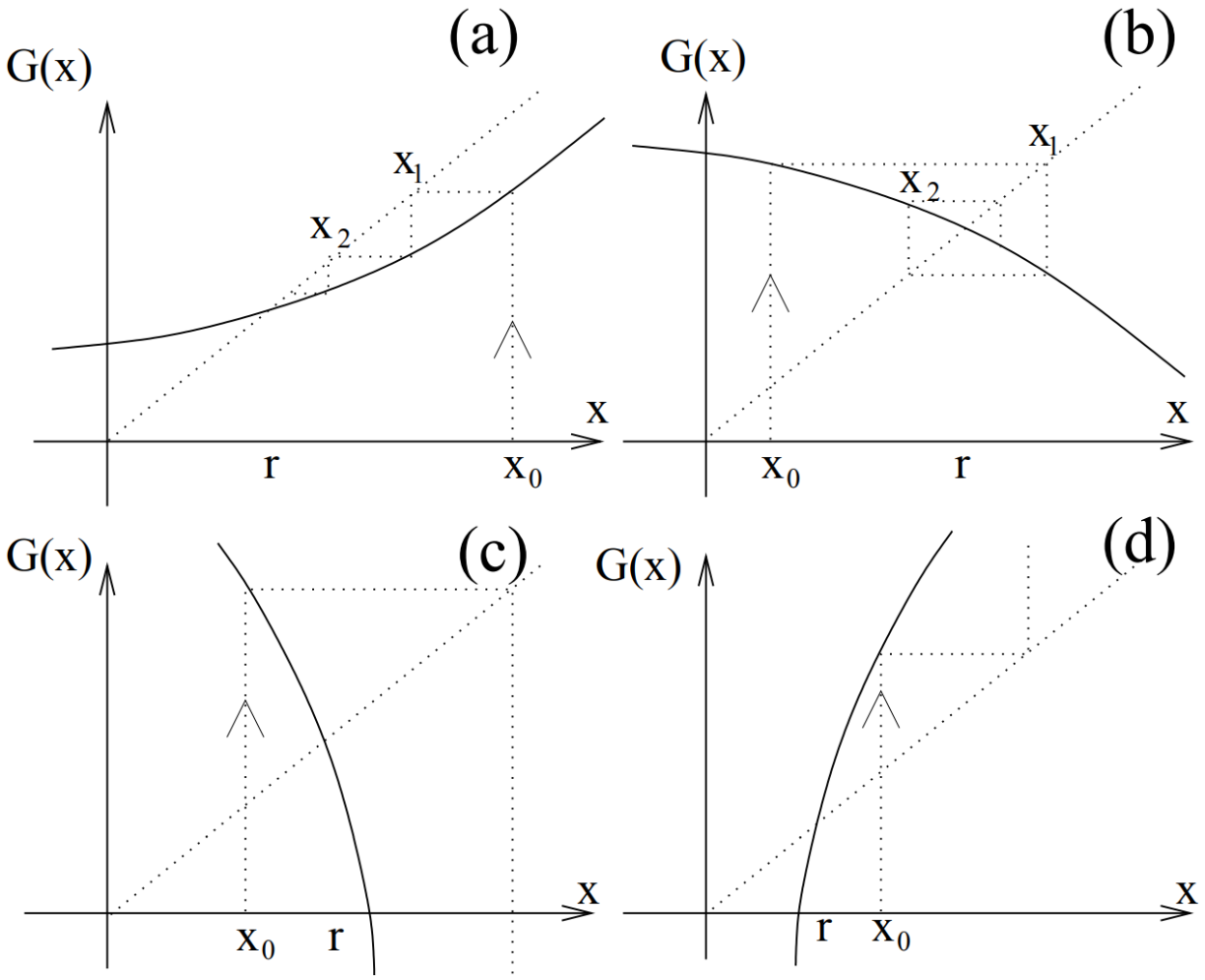
Why does this make sense? Error analysis. Let's look at the error ϵ_n at the nth step. The nth estimate x_n can be written as $x_n = \epsilon_n + r$, and we have:

$$\begin{aligned} \epsilon_{n+1} + r &= g(\epsilon_n + r) \\ &\approx g(r) + \epsilon_n g'(r) \\ \epsilon_{n+1} &\approx \epsilon_n g'(r), \end{aligned}$$

where our friend Taylor saved us once again, and we also used $g(r) = r$. Hence, if $|g'(r)| < 1$ then the sequence (e_n) will converge to 0, and (x_n) to r.

Remark: We can in fact say more than that. If $0 < g'(x) < 1$ then convergence of (x_n) is monotonic, whereas if $-1 < g'(x) < 0$ it is oscillatory (ϵ_n alternates sign, which is obvious in the above equation), and if $\exists x$ s.t. $g'(x) = 0$ then it may be both (e.g. try $g(x) = 3x(1-x)$ and $x_0 = 0.01$).

Fixed-point iteration



(a) monotonic convergence
(b) oscillatory convergence
(c),(d) divergence

[courtesy of Gavin Esler @ DAMTP]

Fixed-point iteration

UNIVERSITY OF CAMBRIDGE

The evolution/beheaviour of (x_n) can be visualized using Cobweb plots:
`https://en.wikipedia.org/wiki/Cobweb_plot`

Live cobweb plots: `https://www.desmos.com/calculator/unan9xh0og`

Golden ratio example. Re-write the positive golden ratio $\phi^2 = \phi + 1$ as the solution to $x = \sqrt{x+1}$, i.e. the fixed point of $g(x) = \sqrt{x+1}$. Letting $x_{n+1} = \sqrt{x_n + 1}$ with $x_0 = 2$ we get:

Fixed-point iteration

UNIVERSITY OF CAMBRIDGE

i	x	err	err/prev. err
1	1.732050807568877 2	1.1402e-01	
2	1.652891650281069 5	3.4858e-02	3.0572e-01
3	1.628769980777233 3	1.0736e-02	3.0800e-01
4	1.621348198499394 9	3.3142e-03	3.0870e-01
5	1.619057811969478 5	1.0238e-03	3.0892e-01
	...		
26	1.618033988749914 7	1.9762e-14	3.0690e-01
27	1.618033988749900 9	5.9952e-15	3.0337e-01
28	1.618033988749896 7	1.7764e-15	2.9630e-01
29	1.618033988749895 3	4.4409e-16	2.5000e-01
30	1.618033988749894 9	0.0000e+00	0.0000e+00
31	1.618033988749894 9	0.0000e+00	nan
32	1.618033988749894 9	0.0000e+00	nan

Fixed-point iteration

UNIVERSITY OF CAMBRIDGE

Analysing the error we get:

$$\begin{aligned}\epsilon_{n+1} &\approx \epsilon_n g'(\phi) \\ &= \epsilon_n \left(2\sqrt{\phi+1}\right)^{-1} \\ &\approx 0.3\epsilon_n\end{aligned}$$

i.e. first-order convergence. Note that this doesn't mean all fixed-point iteration methods have first-order convergence.

Fixed-point iteration

We said that convergence depends on the choice of $g(x)$. What if we chose $g(x) = x^2 - 1$ instead of $g(x) = \sqrt{x+1}$? We have $g'(x) = 2x$. We can easily show that there is no interval $I = [a, b]$ such that we have both $g(I) \subseteq I$ and $|g'(I)| \subset [0, 1)$. So far, this means we are simply *not guaranteed* convergence.

$|2x| < 1$ implies $a > -1/2$ and $b < 1/2$ but $g([-0.5, 0.5]) = [-1, -0.75]$, i.e. there is no $x \in (-0.5, 0.5)$ such that $g(x) = x$. Will it necessarily diverge?

Divergence criterion. If $|g'(r)| > 1$ for all fixed points r of g then r are called *repelling fixed points* and all sequences (x_n) will diverge (unless $x_0 = r$, of course). In practice, if $|g'(x)| \gg 1$ for all x in the vicinity of r then we expect divergence.

Hence, the answer here is yes, the above choice for $g(x)$ will always give divergent sequences (x_n) unless x_0 is ϕ or ψ.

Fixed-point iteration

UNIVERSITY OF CAMBRIDGE

Setting $x_0 = \phi + 10^{-16}$ gives the following (x_n):

i	x	err	err/prev. err
1	1.6180339887498951	2.2204e-16	
2	1.6180339887498958	8.8818e-16	4.0000e+00
3	1.6180339887498978	2.8866e-15	3.2500e+00
4	1.6180339887499045	9.5479e-15	3.3077e+00
5	1.6180339887499260	3.1086e-14	3.2558e+00
6	1.6180339887499957	1.0081e-13	3.2429e+00
7	1.6180339887502213	3.2641e-13	3.2379e+00
	...		
32	3.9828994989829472	2.3649e+00	3.8503e+00
33	14.8634884189986121	1.3245e+01	5.6009e+00
34	219.9232879817058688	2.1831e+02	1.6482e+01

Fixed-point iteration

UNIVERSITY OF CAMBRIDGE

So how do I chose my $g(x)$ and x_0 in practice? One way is to follow the previous approach: Extract scenarios that fall under the convergence criterion. Specifically, analyse the behaviour of $g'(x)$ and of $g(x)$, bounding each of them accordingly, in order to determine a corresponding interval I, then choose any x_0 in that interval. [Intentionally vague to enable supervision exercises.]

Exercise: Find an interval I and a starting x_0 for the other choice of g in the previous exercise to find roots of $f(x) = x^3 - 7x + 2$. Obviously, please present analytic arguments rather than trial and error.

Fixed-point iteration

UNIVERSITY OF CAMBRIDGE

You can also use the fixed-point theory to solve theoretical problems or prove existence of roots and convergent sequences before running the iterations.

Exercise: Let $f : \mathbb{R} \to \mathbb{R}$ be a differentiable function, with $|f'(x)| \leq 0.49$ for all x. Show that the equation $f(x) = \dfrac{2x + \sin(x)}{2}$ has a unique solution in $\mathbb{R}$.

Newton-Raphson method

UNIVERSITY OF CAMBRIDGE

Now that you know a bit about fixed-point iteration, let's introduce the most widely known root finding method, the Newton-Raphson (NR) method. Again, we wish to find roots of $f(x)$ using a converging sequence (x_n). But we want to do it faster. There are several ways to derive the NR expression. We shall call on Taylor, again – we're not as awesome as Newton and Raphson, even though neither of them gave the form that is used today, i.e. using calculus, which was given by Simpson (*).

(*) Newton's original method (1685) was purely algebraic, which he applied only to polynomials and used a sequence of polynomials instead of successive approximations x_n. Raphson's simplified version (1690) was also only algebraic and he applied it only to polynomials but used x_n approximations. Simpson gave the form used today 50 years later (1740), along with other important results in the same paper.

Assuming r is a root of f and that f is continuously differentiable in the vicinity of r with $f'(r) \neq 0$, then a sequence (x_n) that converges to r for $n \to \infty$ can be found using the Taylor expansion of f:

$$f(r) = f(x_n + \epsilon_n) = f(x_n) + f'(x_n)\epsilon_n + O(\epsilon_n^2) = 0.$$

$$\epsilon_n \approx -\frac{f(x_n)}{f'(x_n)}.$$

$$r = x_n + \epsilon_n \approx x_n - \frac{f(x_n)}{f'(x_n)}.$$

in other words $x_n - \frac{f(x_n)}{f'(x_n)}$ is the next estimate of r, and hence we write:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)},$$

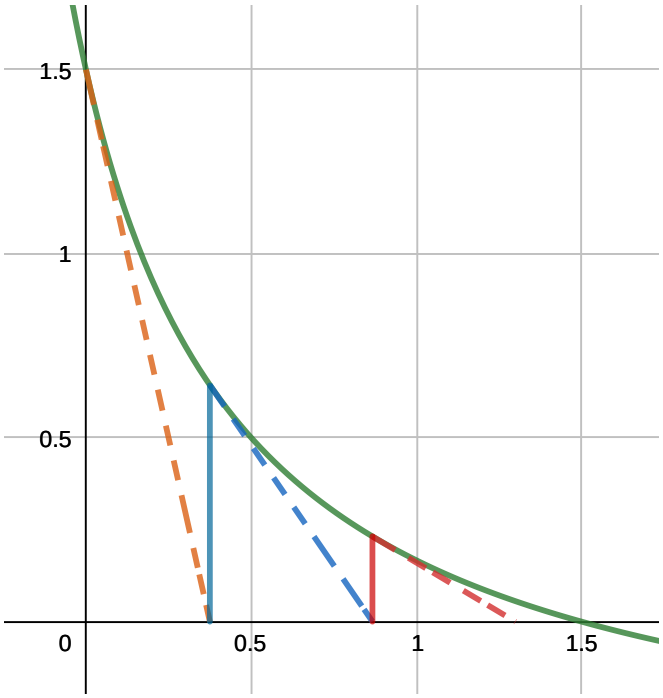
which is the well known NR expression.

It is also a fixed-point iteration method: $g(x) = x - \dfrac{f(x)}{f'(x)}$ and $x_{n+1} = g(x_n)$.

Newton-Raphson method

In short, the NR method improves an initial estimate x_0 of the root by repeatedly setting $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$.

This can be interpreted geometrically as following a tangent to the function to $y = 0$, and using the corresponding x as the next approximation.



Newton-Raphson method

UNIVERSITY OF CAMBRIDGE

Why would you be cool if you regarded NR as a fixed-point iteration method? Because knowing the theory behind fixed-point iteration methods allows you to say more about the NR method. For example, you can extract conditions that guarantee convergence for the NR method.

Exercise: What conditions should f satisfy for the NR method to have guaranteed convergence? Answer: Use the fixed-point convergence criterion for the NR iterator function g.

Newton-Raphson method - Order of convergence

UNIVERSITY OF CAMBRIDGE

What about convergence speed? Let r be the root of $f(x)$ which we hope to converge to. Letting $\epsilon_n = x_n - r$ as usual gives:

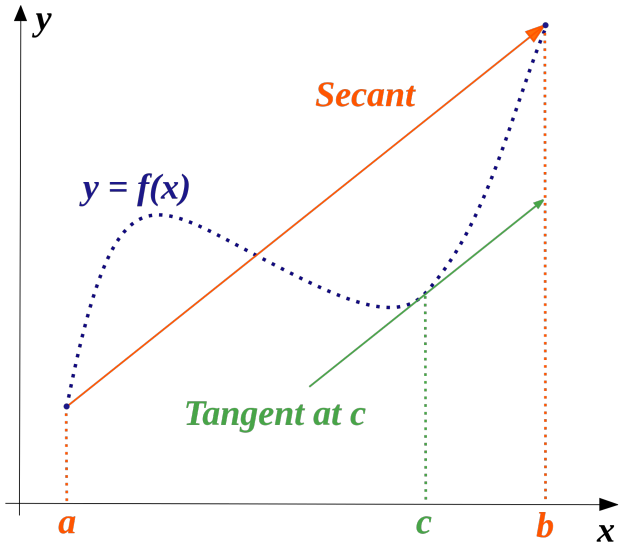
$$\begin{aligned}
\epsilon_{n+1} &= x_{n+1} - r = x_n - \frac{f(x_n)}{f'(x_n)} - r = \epsilon_n - \frac{f(x_n)}{f'(x_n)} \\
&= \epsilon_n - \frac{f(r+\epsilon_n)}{f'(r+\epsilon_n)} \\
&= \epsilon_n - \frac{f(r) + \epsilon_n f'(r) + \epsilon_n^2 f''(r)/2 + O(\epsilon_n^3)}{f'(r) + \epsilon_n f''(r) + O(\epsilon_n^2)} \qquad \text{(Taylor)} \\
&= \epsilon_n - \epsilon_n \frac{f'(r) + \epsilon_n f''(r)/2 + O(\epsilon_n^2)}{f'(r) + \epsilon_n f''(r) + O(\epsilon_n^2)} \qquad \text{(Since } f(r) = 0\text{)} \\
&\approx \epsilon_n^2 \frac{f''(r)}{2f'(r)} + O(\epsilon_n^3) \qquad (\epsilon_n \text{ is small so bottom line} \approx f'(r))
\end{aligned}$$

i.e. second-order convergence. Spot any flaws with the above proof?

Taylor remainder and Error bound

Let's derive the error bound introduced earlier in the course. In many practical applications, we want to obtain an upper bound on the error of our approximations. This is highly desirable, and many times required as part of the design of algorithms, evaluation of their performance or to offer performance guarantees, the latter being vital in some life critical applications.

The following is the derivation done at the blackboard.



From Wikipedia

We first introduce the Mean Value Theorem (without proof). Cauchy's MVT states that given a continuously differentiable function f on an interval $[a, b]$, there exists $c \in [a, b]$ such that

$$f'(c) = \frac{f(b) - f(a)}{b - a}.$$

Geometrically, it means there always exists some point C between any 2 points on a continuously differentiable function, such that the line passing through said 2 points, *the secant*, is parallel to the tangent at C.

However, we can also regard the above secant to be a (gross) approximation of f on $[a, b]$ in the absence of more information about f. Hence, substituting $a = x_0, b = x, c = \xi_1$ we can thus rewrite the above equation as

$$f(x) = f(x_0) + f'(\xi_1)(x - x_0).$$

This is in fact a truncated Taylor series to the 1st term (zero order polynomial), and

$$R_1(x) = f'(\xi_1)(x - x_0)$$

is the remainder of our approximation (first order polynomial). We'd like to find an expression, and in the end a bound, for such a remainder when the Taylor series is truncated to k terms. We now assume that f is $k + 1$ times differentiable around x_0.

Aiming to obtain the next Taylor polynomial approximations, we need to make

the 2nd derivative appear. What if we use the MVT again but this time on the function $f'(x)$? We have

$$f''(\xi_2) = \frac{f'(x) - f'(x_0)}{x - x_0}$$

which rearranged gives

$$f'(x) = f'(x_0) + f''(\xi_2)(x - x_0).$$

Since we want to obtain $f(x)$ we now integrate, and we do so between x_0 and x, i.e. apply $\int_{x_0}^{x} \cdot dt$ (to avoid confusion, write f' using a different variable, e.g. $f'(t) = f'(x_0) + f''(\xi_2)(t - x_0)$)

$$f(x) - f(x_0) = f'(x_0)(x - x_0) + f''(\xi_2)\frac{(x - x_0)^2}{2}.$$

After rearranging, we find the Taylor expansion up to the 2nd term, and the remainder

$$R_2(x) = f''(\xi_2)\frac{(x - x_0)^2}{2}.$$

Obviously, continuing in the same manner up to the nth term, we find that

$$R_{k+1}(x) = f^{(k+1)}(\xi_{k+1})\frac{(x-x_0)^{k+1}}{(k+1)!}.$$

This is the **Lagrange form of the Taylor remainder.** The proof follows by induction as we did above, via MVT and then integration.

Mathematically, bounding $|R|$ in the above expression means to find a constant M that is greater than all values of $|f^{(k+1)}(\xi_{k+1})|$ when ξ_{k+1} ranges in a chosen neighbourhood of the expansion point x_0, typically $[x_0-\alpha, x_0+\alpha]$ for some $\alpha > 0$. We write

$$M = M_{n,\alpha} \geq |f^{(k+1)}(\xi)|$$

for all $\xi \in [x_0-\alpha, x_0+\alpha]$ and thus

$$|R(x)| \;\leq\; M\,\frac{|x-x_0|^{k+1}}{(k+1)!}$$

In practice, in the above expression for $R_{k+1}(x)$, x would be our input value and x_0 our evaluation point, and together they give the error range at the

input, while $R(x)$ is the error at the output sit it is the difference between the true output value $f(x)$ and the approximated value $f^*(x)$ (i.e. the truncated Taylor series). We can thus write

$$|f(x) - f^*(x)| \leq M \frac{|x - x^*|^{k+1}}{(k+1)!}$$

which is called the **Lagrange error bound**. We can compute M more simply as

$$M = \max_{\xi \in [\min(x,x^*), \max(x,x^*)]} |f^{(k+1)}(\xi)|.$$

The above M will give the smallest upper bound on the error, which is desirable in practice. However, it may be that we cannot compute it, e.g. because we can't or it's expensive to compute $f^{(k+1)}(\xi)$. In that case, we can choose any M that is greater than the above expression and still obtain an upper bound; it just won't be the smallest upper bound.

Newton-Raphson method - Order of approximation

Let's now use the Lagrange error bound to investigate the NR method's convergence speed. Obviously, for NR we have $k = 1$. Now set $x = r$ (the root) and $x^* = x_n$ (the nth NR estimate). Hence, denoting the error at the nth iteration by $\epsilon_n = r - x_n$ we have:

$$|f(r) - f^*(r)| \leq \frac{M|r - x_n|^2}{2}$$

$$|f(r) - f(x_n) - \epsilon_n f'(x_n)| \leq \frac{M\epsilon_n^2}{2}.$$

But $f(r) = 0$ and the NR equation gives $x_n - x_{n+1} = \frac{f(x_n)}{f'(x_n)} = \epsilon_{n+1} - \epsilon_n$, hence:

$$|\epsilon_{n+1} f'(x_n) - \epsilon_n f'(x_n) + \epsilon_n f'(x_n)| \leq \frac{M\epsilon_n^2}{2}$$

$$|\epsilon_{n+1}| \leq \frac{M\epsilon_n^2}{2|f'(x_n)|}. \qquad \square$$

You are now *extra* cool, because you have shown that NR has second-order convergence while at the same time providing an upper bound for the error (recall that you can compute M).

Conditions: Remember that the above is subject to several conditions on f' and f'', as specified in the NR method and the Lagrange remainder; specifically: $f'(x) \neq 0$ in the vicinity of r, $f''(x)$ continuous in the vicinity of r. We may also add that x_0 must be sufficiently close to r to avoid divergence.

Newton-Raphson method - Golden ratio example

UNIVERSITY OF CAMBRIDGE

To compute the golden ratio, we have $f(x) = x^2 - x - 1$.

i	x	err	err/prev. err
0	2.0	3.8197E-01	
1	1.66666666666667	4.8633E-02	1.2732E-01
2	1.61904761904762	1.0136E-03	2.0843E-02
3	1.61803444782168	4.5907E-07	4.5290E-04
4	1.61803398874999	9.4147E-14	2.0508E-07
5	1.61803398874990	0.0000E+00	0.0000E+00

Second-order convergence is much faster: it requires a number of steps proportional to the logarithm of the desired number of digits.

Exercise: Use the NR method to find the golden ratio as a fixed-point iteration method, by considering the iteration function $g(x) = x - f(x)/f'(x)$:

`https://www.desmos.com/calculator/unan9xh0og`

Newton-Raphson method

UNIVERSITY OF CAMBRIDGE

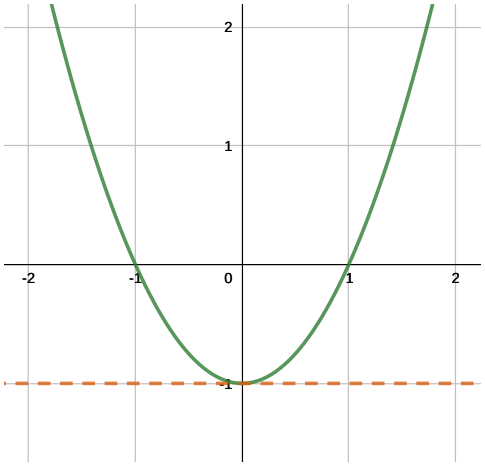
How to choose x_0? This is one of the most important steps. A "bad" initial guess x_0 can cause divergence. A common approach to find a "good" x_0 is to first linearise the function f, i.e. a linear approximation of f, and then extract x_0 as the root of that. For example, let $f(x) = x\ \sin(\pi x) + e^{-x}$. We can linearise this using Taylor, truncate it to the linear term, which is $f(x) \approx 1 - x$ and extract x_0 as its root, i.e. $x_0 = 1$. Note that in practice you may not always be able to use Taylor to linearise, and you may need to resort to other techniques (logs, differentials, evolutionary algorithms etc – beyond the course, but the curious can look these up).

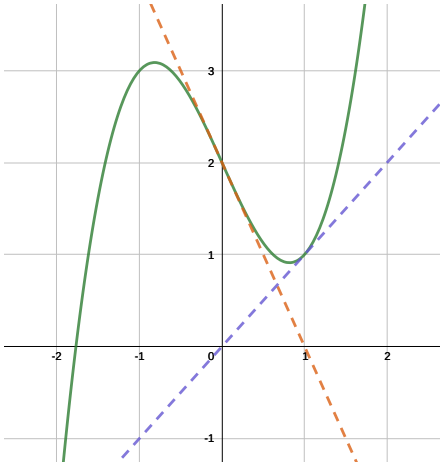
Newton-Raphson method

UNIVERSITY OF CAMBRIDGE

The NR method usually converges quickly, but does have known problems:

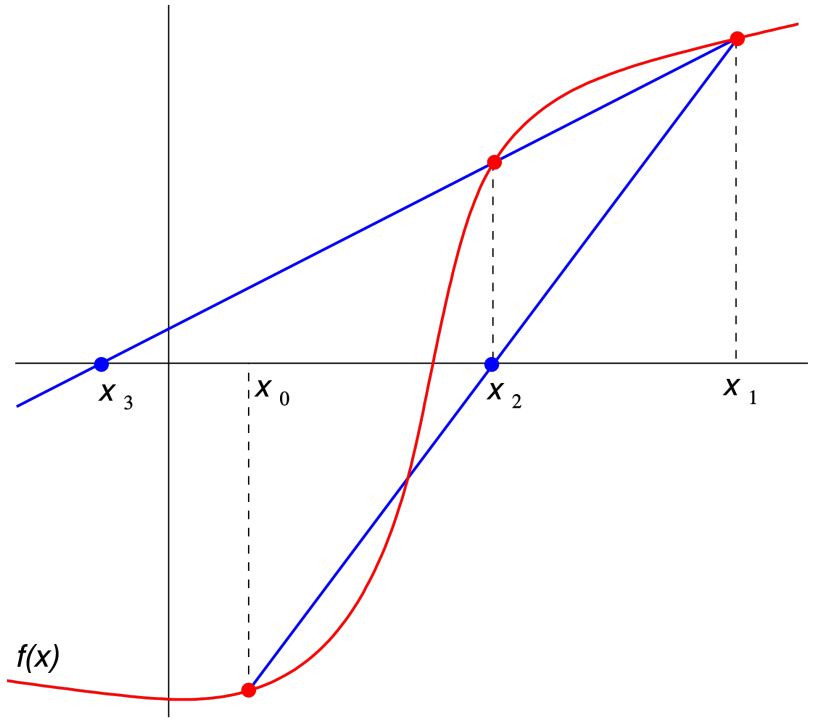
- *Overshoot.* It overshoots if the 1st derivative is badly behaved at r. For example, NR for $f(x) = |x|^a$ for $a \in [0, 1/2]$ will overshoot and diverge, as $f'(x)$ is very large near the root $x = 0$ (where it tends to infinity). For $a = 1/2$ it will oscillate between two values.
- *Only linear (first-order) convergence for roots with multiplicity.* However, if the multiplicity m of the root is known then the modified algorithm $x_{n+1} = x_n - m\frac{f(x_n)}{f'(x_n)}$ preserves quadratic convergence.
- *Bad initial guess x_0.* It can cause divergence but there are workarounds as explained earlier.





- *Stationary points.* If any x_n is stationary point then this will throw the sequence off and it will diverge (division by 0, essentially).
- *Possibility of oscillating.* For example, $f(x) = x^3 - 2x + 2$. Choosing $x_0 = 0$ will oscillate between $x_{2n} = 0$ and $x_{2n+1} = 1$. What if we extracted a "better" x_0 as explained earlier?

Secant method



From Wikipedia

Start with two initial values x_0 and x_1 (any two points?). Build the line through the points $(x_0, f(x_0))$ and $(x_1, f(x_1))$:

$$y = \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_1) + f(x_1).$$

Find the root of this line:

$$x = x_1 - f(x_1)\frac{x_1 - x_0}{f(x_1) - f(x_0)},$$

which becomes the next estimate x_2.

Secant method

UNIVERSITY OF CAMBRIDGE

Repeat with x_1 and x_2 and so on, to obtain the general recurence for the secant method:

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

or:

$$x_{n+1} = \frac{x_{n-1}f(x_n) - x_n f(x_{n-1})}{f(x_n) - f(x_{n-1})}$$

Secant method

UNIVERSITY OF CAMBRIDGE

It can be regarded as a result of the NR method. Recall the numerical differentiation approximation? We can say that each secant line is in fact an approximation of the derivative at the new point, i.e.

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}.$$

Substitute this into the NR equation $x_{n+1} = x_n - \dfrac{f(x_n)}{f'(x_n)}$ and we obtain the secant method equation.

In other words, in the NR method, if instead of $f'(x_n)$ you compute a numerical approximation of the derivative at x_n, then you are in fact doing the secant method, and not the NR method.

Secant method

UNIVERSITY OF CAMBRIDGE

How fast does it converge? Let's derive its **order of convergence**. We'll try to find the value of p such that $|\epsilon_{n+1}| \approx |\epsilon_n|^p$. Let $x_n = \epsilon_n + r$ where $f(r) = 0$, i.e. r is a root. Substituting in the secant recurrence we get:

$$\epsilon_{n+1} = \epsilon_n - f(x_n)\frac{\epsilon_n - \epsilon_{n-1}}{f(x_n) - f(x_{n-1})}$$

$$= \frac{\epsilon_{n-1}f(x_n) - \epsilon_n f(x_{n-1})}{f(x_n) - f(x_{n-1})}$$

$$= \epsilon_n\epsilon_{n-1}\frac{\frac{f(x_n)}{\epsilon_n} - \frac{f(x_{n-1})}{\epsilon_{n-1}}}{f(x_n) - f(x_{n-1})}$$

$$= \epsilon_n\epsilon_{n-1}\frac{\frac{f(x_n)-f(r)}{x_n-r} - \frac{f(x_{n-1})-f(r)}{x_{n-1}-r}}{f(x_n) - f(x_{n-1})}$$

knowing that subtracting $f(r)$ doesn't change anything. Denoting by $F(x) = \frac{f(x)-f(r)}{x-r}$ then we have

$$\epsilon_{n+1} = \epsilon_n \epsilon_{n-1} \frac{F(x_n) - F(x_{n-1})}{f(x_n) - f(x_{n-1})}.$$

We aim to get rid of the f, F terms. When we see successive terms subtracted, that's an indication that the mean value theorem may come in handy. The mean value theorem says that there exists ν_n, ξ_n between x_n and x_{n-1} such that:

$$f(x_n) - f(x_n - 1) = f'(\nu_n)(x_n - x_{n-1})$$

and

$$F(x_n) - F(x_n - 1) = F'(\xi_n)(x_n - x_{n-1})$$

which gives

$$\epsilon_{n+1} = \epsilon_n \epsilon_{n-1} \frac{F'(\xi_n)}{f'(\nu_n)}.$$

We now need to bound the fraction in order to extract a dependency between

only ϵ's. We only need to find a bound on the interval between x_{n-1} and x_n, and we could already bound it as it is, making an argument that F approximates f' by the way it's defined, and hence F' approximates f'', and so we can say that $F'(\xi_n)/f'(\nu_n)$ is upper bounded over the interval between x_{n-1} and x_n by a constant M corresponding to where F' (and thus f'') is maximum and f' is minimum. However, we can obtain those expressions, by first looking at F':

$$F'(x) = \frac{f(r) + f'(x)(x-r) - f(x)}{(x-r)^2},$$

where we can spot that the numerator is the difference between $f(r)$ and its truncated Taylor expansion up the 1st derivative, which is equal to the Taylor remainder, for which we already derived an expression earlier in the course (reminder: $f(r) = f(x) + f'(x)(r-x) + f''(\zeta)(r-x)^2/2$ where ζ is some value between r and x). Substituting into F' and then back above we get:

$$\epsilon_{n+1} = \epsilon_n \epsilon_{n-1} \frac{f''(\zeta_n)}{2f'(\nu_n)},$$

where we now have f'' explicitly, and we proceed to bounding the fraction as described above, to say that:

$$\epsilon_{n+1} \leq \epsilon_n \epsilon_{n-1} M,$$

in other words

$$\epsilon_{n+1} \approx \epsilon_n \epsilon_{n-1}.$$

Note that for the above bound and relationship to hold we need f to be twice differentiable over the interval of convergence.

The order of convergence is given by p such that $\epsilon_{n+1} \approx \epsilon_n^p$. Remember that all these equation hold in the limit (i.e. when $n \to \infty$). Substituting this above twice, we get:

$$\epsilon_{n-1}^{p^2} \approx \epsilon_{n-1}^{p+1}$$

or $\epsilon_{n-1}^{p^2-p-1} \approx 1$, which means $p^2 - p - 1$ must be 0 (well, tends to 0) as $n \to \infty$ since we know $\epsilon_{n-1} \to 0$. This is the known golden ratio equation, and hence $p = \phi = (1 + \sqrt{5})/2$, i.e. **Superlinear convergence!** Faster than bisection. Slower than NR.

Secant method

UNIVERSITY OF CAMBRIDGE

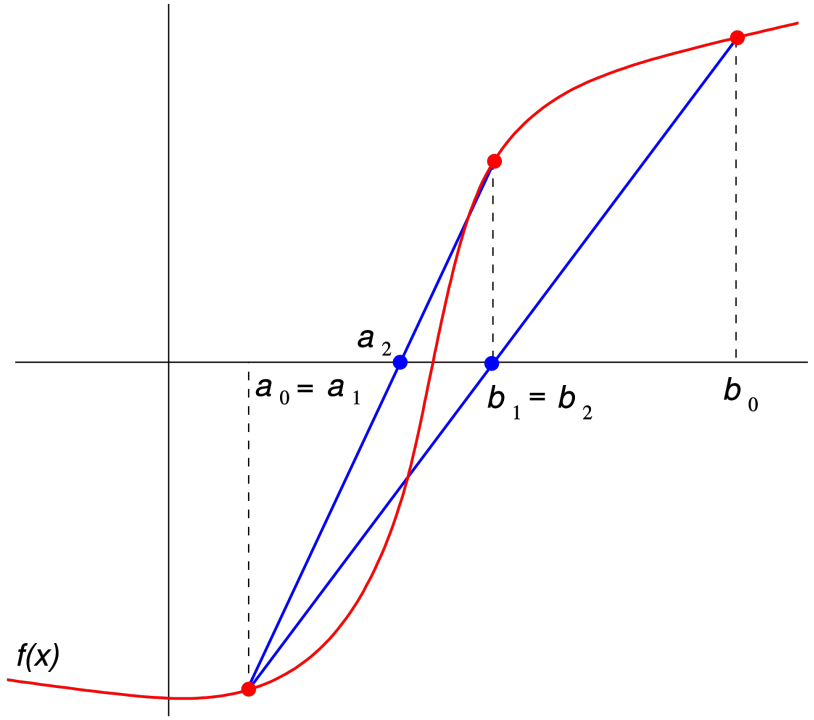
Pros?

- Doesn't need derivatives (can be used where NR cannot).
- Doesn't need to check for opposite signs (hmm...).
- Faster than bisection.
- "Old school" – predates NR by about 3000 years!

Disadvantages?

- Doesn't always converge ... because it doesn't check for opposite signs! It does not enforce bracketing of the root like the bisection method does, so it can fail. But we can modify it so that it does always converge :).

False position method



From Wikipedia

Modified secant method to enforce bracketing of the root. Same recurrence formula as secant, but instead of applying it on x_n and x_{n-1}, it applies it on x_n and the last estimate x_k such that $f(x_n)$ and $f(x_k)$ have different sign. Needs a little storage memory.

Always converges! Provided we start with two points with different signs ...

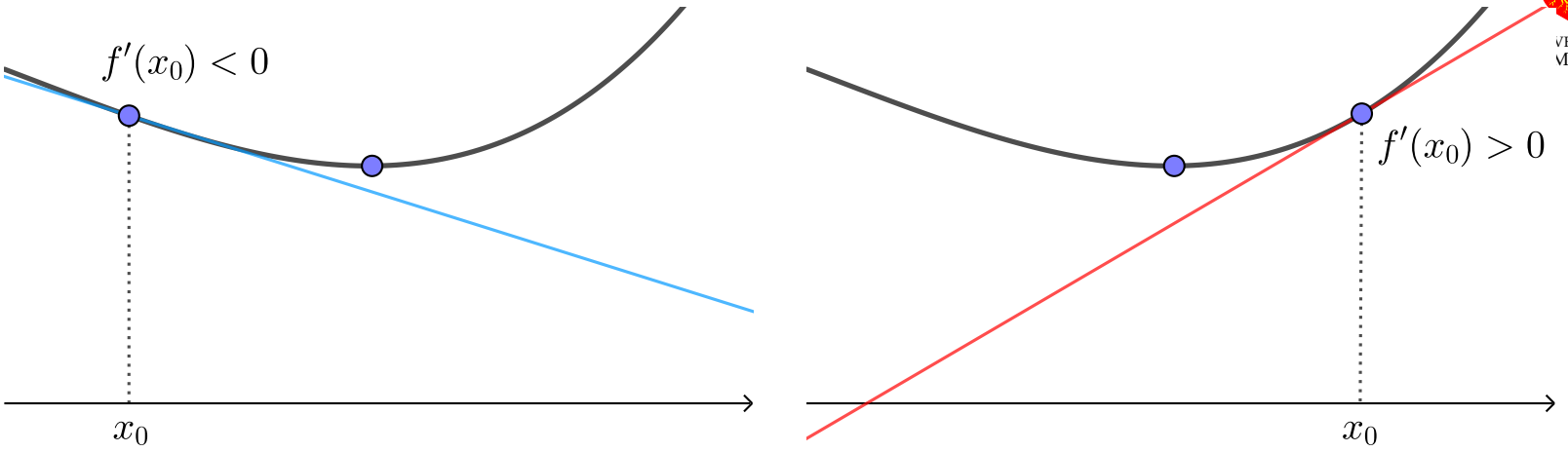
Gradient descent

UNIVERSITY OF CAMBRIDGE

Gradient descent or *steepest* descent is a method used to find minima of a multidimensional function f. Gradient descent is a fundamental concept and method for a huge number of applications, e.g. machine learning, solving linear systems, etc.

Problem: Let $f : \mathbb{R}^N \to \mathbb{R}$ be a continuously differentiable function. We want to find a minimum of f iteratively, using successive estimates $\mathbf{x_n}$, i.e. we want to decrease towards the minimum at every iteration, i.e. $f(\mathbf{x}_{n+1}) \leq f(\mathbf{x}_n)$.

Let's look at the uni-dimensional case ($N = 1$) first.



If we subtract from x_0 a small fraction of $f'(x_0)$, then we are guaranteed to progress towards the minimum. In other words, we should proceed in the opposite direction of the gradient.

Mathematically, if we take $x_1 = x_0 - \gamma f'(x_0)$ where $\gamma > 0$ is *sufficiently small*, then $f(x_1) \leq f(x_0)$. Exercise: Prove that this is true. We then proceed iteratively as

$$x_{n+1} = x_n - \gamma f'(x_n)$$

until $|f'(x_n)| < \epsilon$ for a desired ϵ. Spot the fixed-iteration?

This is yet another fixed-point iteration, with the iterator function

$$g(x) = x - \gamma f'(x),$$

which is in fact not really surprising. As we said earlier, many problems can be recast as root finding problems. Finding a minimum of f can be posed as finding a root of f'. Fixed-point iteration theory now also gives us convergence and divergence criteria.

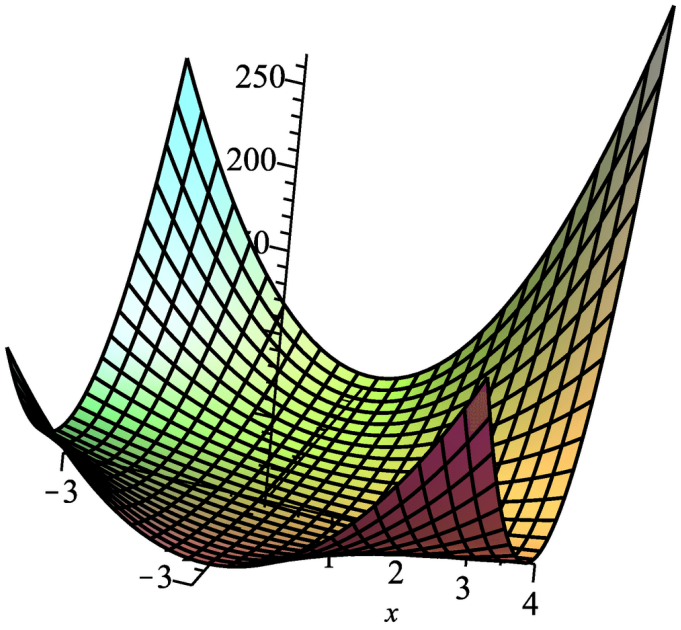
Exercise: Find a minimum point of $f(x) = x^4 - 3x^3 + 5x + 1$. Discuss the importance of "sufficiently small" when choosing γ. *Hint:* Desmos :).

Intuition: As we approach the minimum, $f'(x)$ itself gets smaller and aids in taking gradually smaller steps to avoid overshooting. A sufficiently small γ is important in the early steps, or else we may overshoot the minimum.

NOTE: In machine learning, γ is called the *learning rate*.

Let's now tackle the multidimensional case.

When f is multi-variable we need to be more careful, but we can also do smarter things! When f was uni-dimensional, decreasing towards $-f'(x_0)$ would always hit the minimum (there's nowhere else to go in 1D). However, when f is multidimensional, we have a gradient *vector* ∇f. Decreasing in the direction of $-\nabla f(\mathbf{x}_0)$ will miss the minimum, unless f is symmetric (i.e. the gradient at every point points towards the minimum) or we just happened to be lucky when choosing x_0.



Most functions are not symmetric, which means we will have to change direction at every step. The same principle holds, though: Search in the direction opposite the gradient. In the absence of other information about f, this would be the *steepest* descent (*), hence the name.

The fact that the minimum is not (guaranteed to be) in the direction of $-\nabla f(\mathbf{x}_0)$ means there is scope for being smarter regarding γ: it allows us to compute γ adaptively, i.e. we can choose between a fixed γ or computing γ_n at every iteration which can speed up the process. The algorithm is thus as follows, starting with a point $\mathbf{x}_0$:

1. Set the search direction $\mathbf{d}_n = -\nabla f(\mathbf{x}_n)$.
2. Compute γ_n or use a fixed (but small) $\gamma_n = \gamma$.
3. Set $\mathbf{x}_{n+1} = \mathbf{x}_n + \gamma_n \mathbf{d}_n$.
4. Repeat until $||\nabla f(\mathbf{x}_n)|| < \epsilon$ for a desired ϵ.

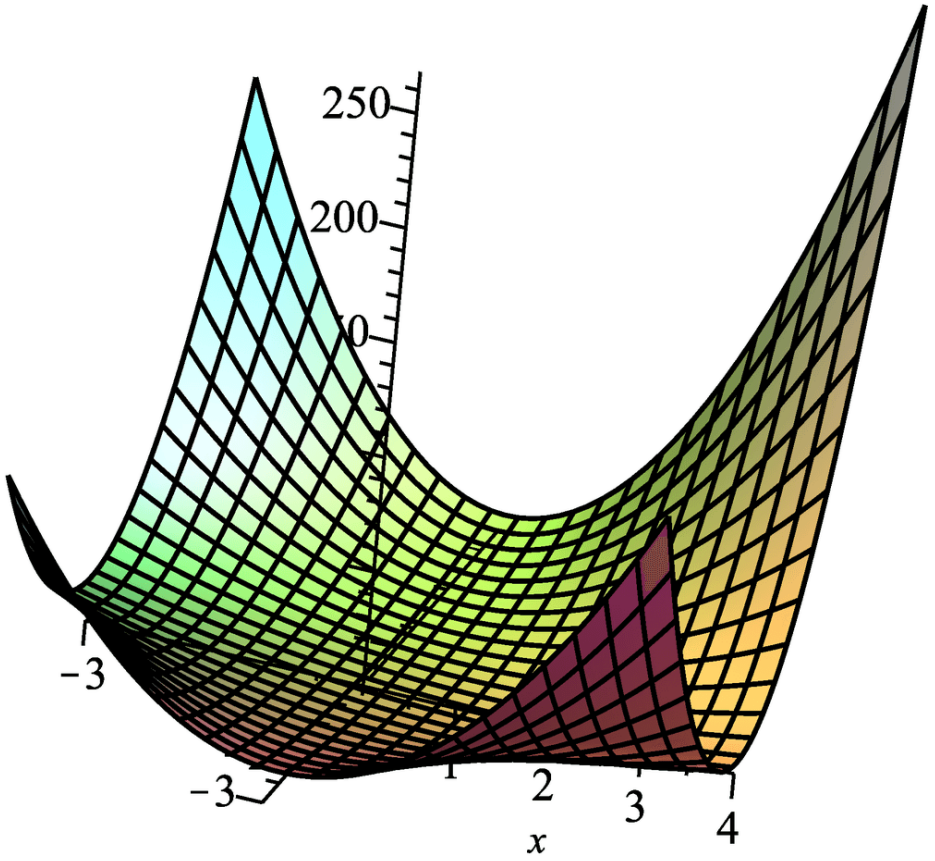
(*) Not necessarily fastest convergence! Lookup *conjugate gradient descent*.

Gradient descent - fixed γ

Let $f(x,y) = x^2y^2 + x^2 + y^2 - 3x - 2y$, $\gamma = 0.1$ be fixed, and $\mathbf{x}_0 = [0,0]^T$.

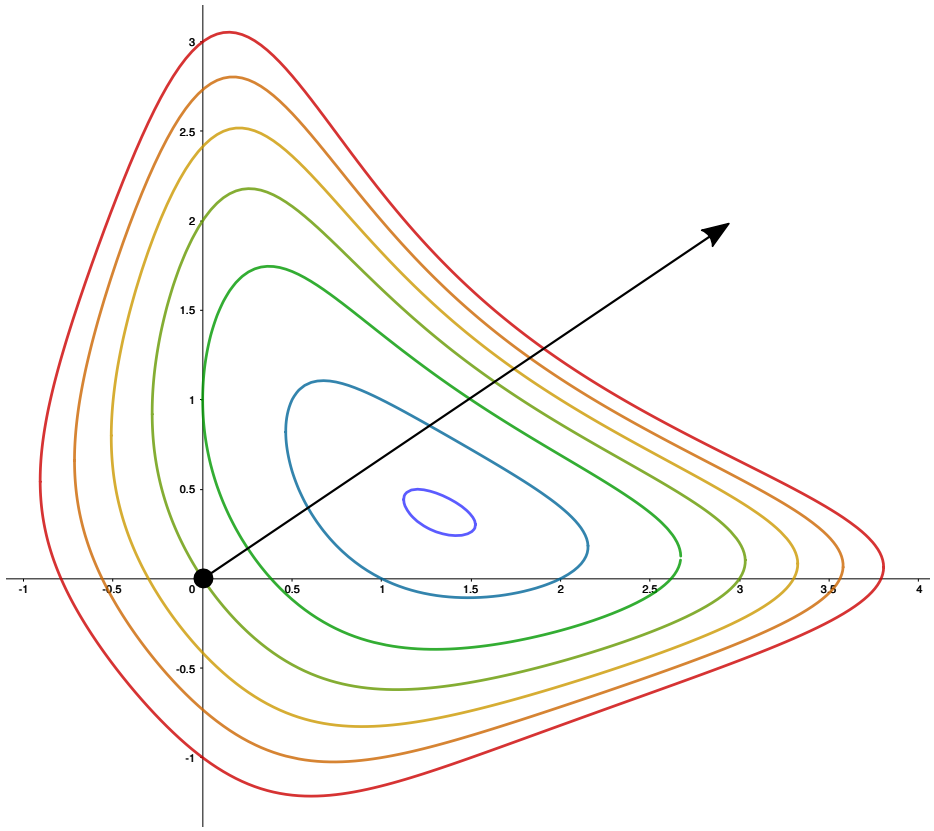
$$\nabla f(\mathbf{x}) = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix}$$

$$= \begin{pmatrix} 2xy^2 + 2x - 3 \\ 2x^2y + 2y - 2 \end{pmatrix}$$



Gradient descent - fixed γ

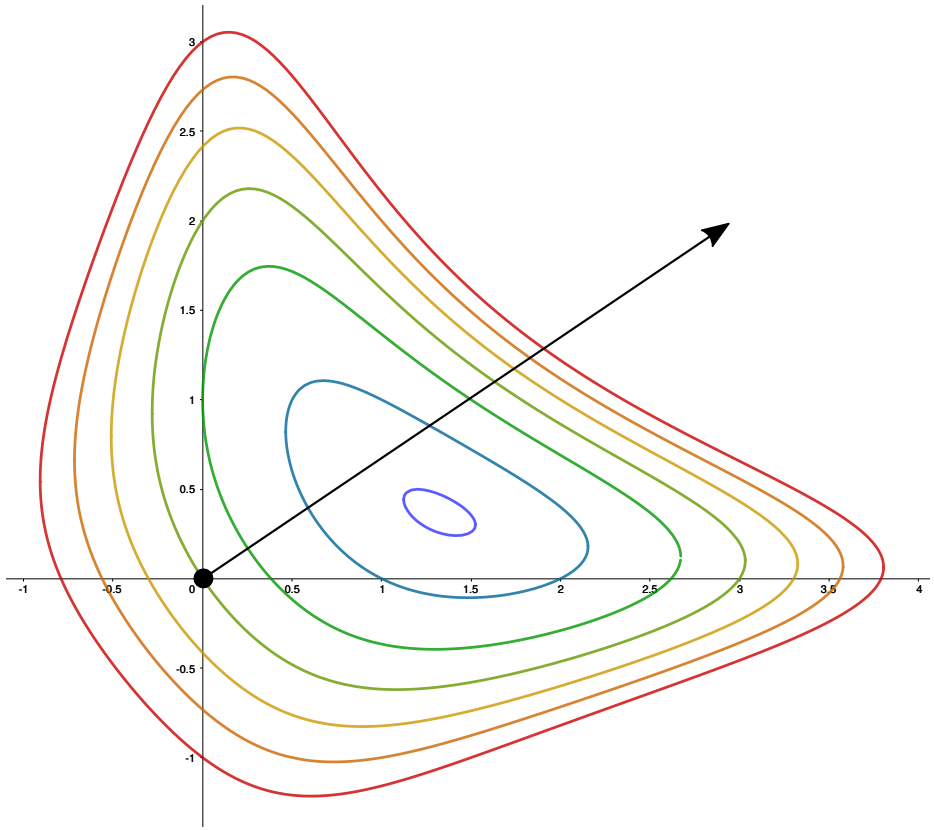
$$\mathbf{x}_0 = [0, 0]^T$$
$$\nabla f(\mathbf{x}_0) = [-3, -2]^T$$
$$\mathbf{x}_1 = \mathbf{x}_0 - 0.1\nabla f(\mathbf{x}_0)$$
$$= [0.3, 0.2]$$



Gradient descent - adaptive γ

$$\mathbf{x}_0 = [0, 0]^T$$

$$\nabla f(\mathbf{x}_0) = [-3, -2]^T$$



Gradient descent - adaptive γ

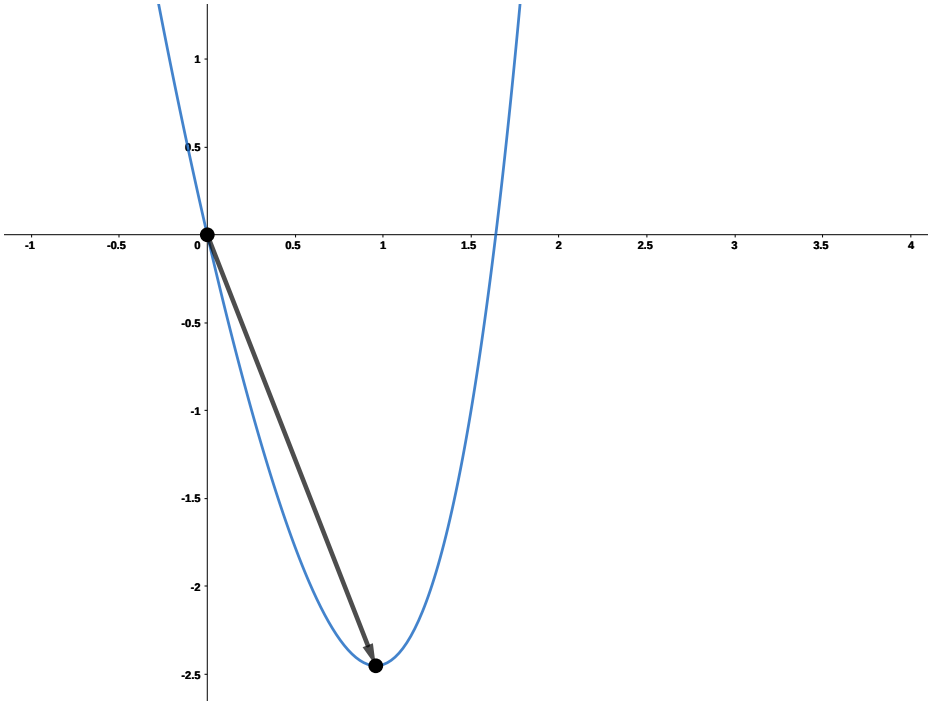
UNIVERSITY OF CAMBRIDGE

Find γ_n that minimises the projection of f onto the plane given by $\nabla f(\mathbf{x}_n)$, i.e. find γ_n as the solution to $\frac{d}{d\gamma} f(\mathbf{x}_{n+1}) = \frac{d}{d\gamma} f(\mathbf{x}_n - \gamma \nabla f(\mathbf{x}_n)) = 0$.

$$\mathbf{x}_0 = [0, 0]^T$$
$$\nabla f(\mathbf{x}_0) = [-3, -2]^T$$

Project f onto the plane $y = 2/3x$ to obtain a 1D function $f(x, 2/3x) = 4x^4/9 + 13x^2/9 - 13x/3$. Its minimum is at $x_0 = 0.96$ (how?), which gives $y = 0.64$. Hence

$$\begin{aligned}\mathbf{x}_1 &= \mathbf{x}_0 + [0.96, 0.64]^T \\ &= [0.96, 0.64]^T.\end{aligned}$$

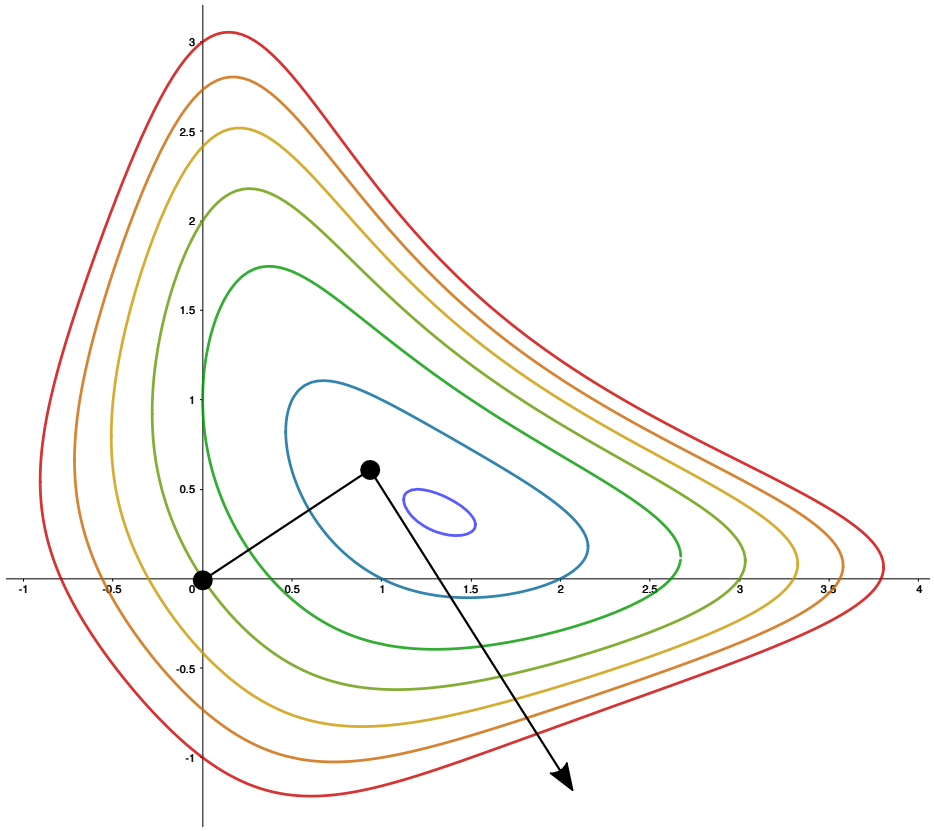


Gradient descent - adaptive γ

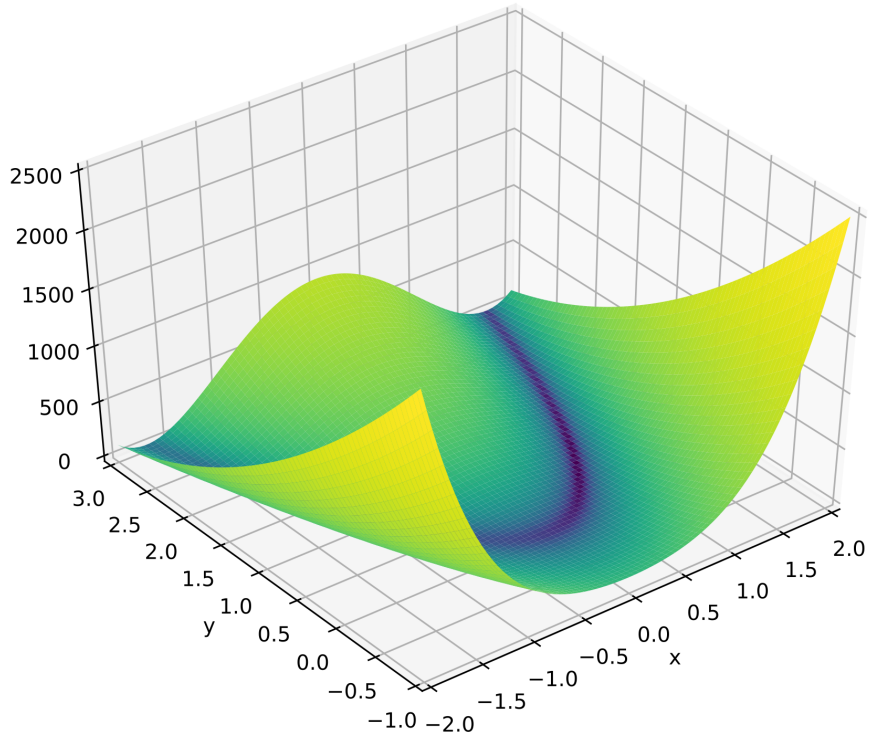
$$\mathbf{x}_1 = [0.96, 0.64]^T$$

$$\nabla f(\mathbf{x}_1) = [-0.29, 0.45]^T$$

Repeat.



Gradient descent



From Wikipedia

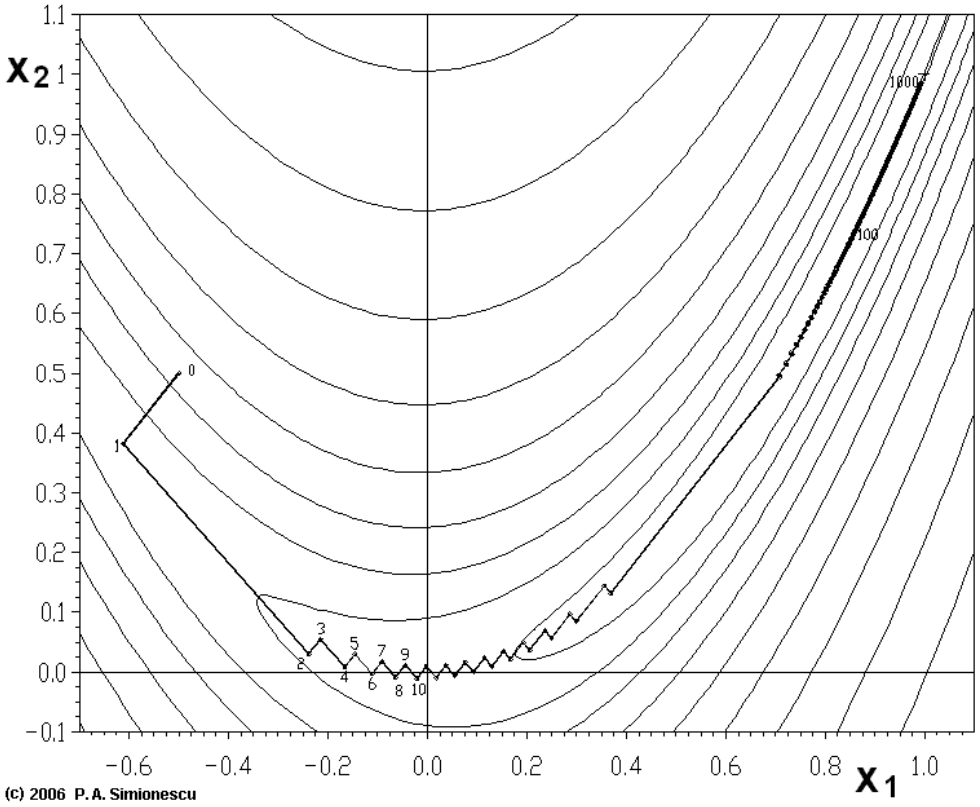
As with the NR method, it has undesired behaviour if the gradient has unhelpful properties.

e.g. the Rosenbrock function:

$$f(x, y) = (1 - x)^2 + \\ + 100(y - x^2)^2$$

Narrow curved valley which contains the minimum. The bottom of the valley is very flat.

Gradient descent



From Wikipedia

Rosenbrock function:

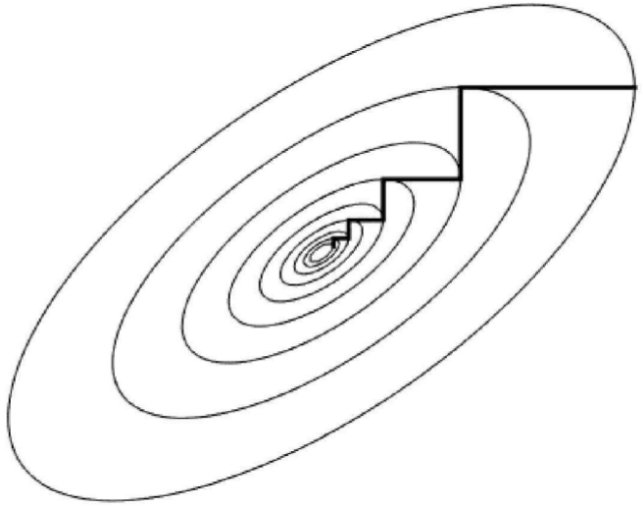
$$f(x,y) = (1-x)^2 + \\ + 100(y-x^2)^2$$

Because of the curved flat valley the gradient descent is zig-zagging very slowly with small step sizes towards the minimum. It's repeating the same search direction (i.e. parallel or close to parallel vectors) many times.

Gradient descent

UNIVERSITY OF CAMBRIDGE

The worst case for gradient descent is when each gradient line at one point is the tangent line at the next point (valley), repeating the same search direction many times:



Local minima only: As was obvious, gradient descent only finds local minima, dictated by the choice of $\mathbf{x}_0$. It finds the global minimum only if f is convex.

Simulated annealing

UNIVERSITY OF CAMBRIDGE

Simulated annealing is a Monte Carlo method for finding the global minimum (or maximum) of a function. It may be preferred over gradient descent and its relatives when finding an approximate global minimum is more important than finding a local minimum precisely.

In each iteration, a new candidate state is selected randomly, and then accepted or discarded randomly.

Simulated annealing

UNIVERSITY OF CAMBRIDGE

The method uses randomness to allow it to regress to worse solutions (with low probability), making it possible to escape from local minima.

The randomness is controlled by the **temperature**, T, with a high temperature meaning more randomness. A high temperature allows more-distant candidate states to be considered, and increases the probability of accepting a worse solution. Over time, the temperature is reduced, allowing the solution to converge to a minimum.

NOTE: Any problem whose state can be evaluated can be solved using this method, e.g. travelling salesman

Simulated annealing

UNIVERSITY OF CAMBRIDGE

Starting with an initial estimate x_0 and T_0:

1. Randomly select a candidate state $c_{n+1} = C(x_n, T_n)$.
2. $x_{n+1} = c_{n+1}$ with probability $P(f(x_n), f(c_{n+1}), T)$, otherwise $x_{n+1} = x_n$.
3. Decrease temperature $T_{n+1} = A(T_n)$.

The choice of C, P, A depend on the problem, but may look something like this:

$$\begin{aligned} C(x,T) &= x + \text{uniform\_random}(-1,1) \cdot e^T \\ P(y,y',T) &= \frac{1}{1+e^{\frac{y'-y}{T}}} \\ A(T) &= 0.999T \end{aligned}$$

A necessary condition is that when $T \to 0$ then $P(y, y', T) \to 0$ if $y' > y$, else to a positive value. Q: Why not always switch to c_{n+1} if $f(c_{n+1}) < f(x_n)$?

Linear Systems

Using matrices to solve simultaneous equations

UNIVERSITY OF CAMBRIDGE

A system of linear simultaneous equations can be written in matrix form.

$$\begin{aligned} x + 2y + 3z &= 1 \\ 4x + 5y + 6z &= 1 \\ 7x + 8y + 8z &= 0 \end{aligned} \qquad \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 8 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

We wish to find $\mathbf{x}$ such that $A\mathbf{x} = \mathbf{b}$.

These kinds of systems appear virtually everywhere in the real world ...

Vectors and matrices - summary of notations/concepts

UNIVERSITY OF CAMBRIDGE

Lowercase bold letters denote vectors, e.g. $\mathbf{x}, \mathbf{b}$, while uppercase bold letters denote matrices, e.g. $\mathbf{A}$.

We may drop the bold face if the context implies it.

We write $A \in \mathbb{R}^{m \times n}$ to denote a real valued matrix with m rows and n columns.

We write A^T to mean the transpose of A, i.e. the ith row becomes the ith column (or vice versa).

$$A = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}, \quad A^T = \begin{pmatrix} 0 & 2 \\ 1 & 3 \end{pmatrix}$$

Transposing gives several useful properties, such as $(A+B)^T = A^T + B^T, (AB)^T = B^T A^T$ and others (more below).

A vector $\mathbf{v}$ with n entries means $\mathbf{v} = [v_1, v_2, ..., v_n]^T$, i.e. a column vector by default, unless otherwise specified. $\mathbf{v}^T$ is a row vector, $\mathbf{v}^T = [v_0, v_1, ..., v_{n-1}]$.

The ℓ^p-**norm** of a vector $\mathbf{v}$ is:

$$\|\mathbf{v}\|_p = (v_1^p + v_2^p + ... + v_n^p)^{1/p}.$$

When $p = 2$ we obtain the **"2-norm"**, or **Euclidean norm**, or **magnitude** and we write $\|\mathbf{v}\|$ for short. The ℓ^2 norm has direct physical meaning (e.g. $\|\mathbf{v}\|^2$ is the discrete signal's energy).

We often work with $\|\mathbf{v}\|^2$ instead, to avoid roots:

$$\|\mathbf{v}\|^2 = \mathbf{v}^T\mathbf{v} = \sum_{i=1}^{n} v_i^2.$$

The **scalar** (or **dot**) **product** of two vectors (separated by angle θ):

$$\mathbf{v} \cdot \mathbf{w} = \mathbf{v}^T\mathbf{w} = \sum_{i=1}^{n} v_i w_i = \|\mathbf{v}\|\|\mathbf{w}\| \cos\theta$$

Two vectors are **orthogonal** if their scalar product is zero.

A set of orthogonal vectors is a set of vectors orthogonal on each other.

A set of **orthonormal** vectors is a set of orthogonal vectors of magnitude 1.

A matrix A is symmetric if $A^T = A$.

A matrix A is anti-symmetric if $A^T = -A$.

A matrix A is diagonal if it's zero everywhere except on the main diagonal. Usually refers to square matrices, which are then also symmetric.

A matrix is **orthogonal** if all of its column vectors are ortho*normal*. Orthogonal matrices have some interesting properties: $AA^T = I$, $\|A\mathbf{v}\|^2 = \|\mathbf{v}\|^2$ and $A\mathbf{v} \cdot A\mathbf{w} = \mathbf{v} \cdot \mathbf{w}$.

The **inverse** of a matrix, A^{-1}, is such that $AA^{-1} = A^{-1}A = I$. The concept of an inverse exists for any sized matrix (does not have to be square). We also have $(AB^{-1})^T = \mathbf{A^T}^{-1}$.

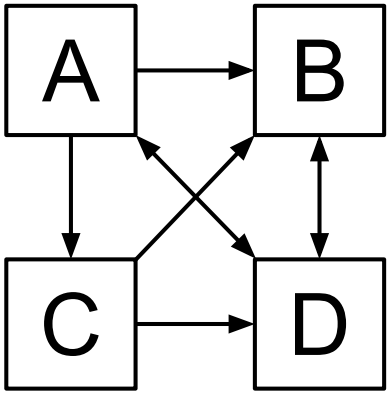
A square matrix is called **non-singular** if it has an inverse.

The **determinant** of a square matrix A, denoted by $|A|$ or $\det(A)$, is a scalar value which encodes useful properties of a matrix. For example, A^{-1} exists iff $|A| \neq 0$; A is non-singular iff $|A| \neq 0$; $|A^T| = |A|$; and others.

Using matrices to solve simultaneous equations

Example: Google's PageRank.

Consider a web page *important* if other important pages link to it.
More-important pages will appear higher in the search results. Each page shares its own importance equally among all pages it links to.



$$\begin{aligned} \frac{1}{2}D &= A \\ \frac{1}{3}A + \frac{1}{2}C + \frac{1}{2}D &= B \\ \frac{1}{3}A &= C \\ \frac{1}{3}A + B + \frac{1}{2}C &= D \end{aligned} \qquad \begin{pmatrix} 0 & 0 & 0 & \frac{1}{2} \\ \frac{1}{3} & 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{3} & 1 & \frac{1}{2} & 0 \end{pmatrix} \begin{pmatrix} A \\ B \\ C \\ D \end{pmatrix} = \begin{pmatrix} A \\ B \\ C \\ D \end{pmatrix}$$

Now solve $M\mathbf{x} = \mathbf{x}$, or equivalently, $(M - I)\mathbf{x} = \mathbf{0}$, where I is the identity matrix.

Using matrices to solve simultaneous equations

UNIVERSITY OF CAMBRIDGE

We will cover a few ways to solve problems of the form $A\mathbf{x} = \mathbf{b}$:

- Matrix inverse
- Gaussian elimination
- LU factorisation
- Cholesky factorisation
- QR factorisation

Each method has its own advantages and disadvantages.

We shall focus for now only on the case when A is square. A unique solution exists *if and only if* A is non-singular, i.e. $\det(A) \neq 0$.

Matrix inverse

UNIVERSITY OF CAMBRIDGE

If we know the matrix A's inverse, A^{-1}:

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ A^{-1}A\mathbf{x} &= A^{-1}\mathbf{b} \\ \mathbf{x} &= A^{-1}\mathbf{b} \end{aligned}$$

but finding the inverse of large matrices can fail or be unstable.

Gaussian elimination

UNIVERSITY OF CAMBRIDGE

This matrix phrasing of the school technique for solving simultaneous equations.

We can freely add multiples of any row to any other (must do so on $\mathbf{b}$ as well):

1. Transform the matrix into **upper-triangular form**:

   For each row $\mathbf{a}_i$, with i from 1 to $n-1$ inclusive:

   For each element $a_{i,j}$, with j from 0 to $i-1$ inclusive:

   Set $a_{i,j}$ to zero by setting $\mathbf{a}_i = \mathbf{a}_i - \frac{a_{i,j}}{a_{j,j}}\mathbf{a}_j$

2. Back substitute to find the unknown values

Upper-triangular: $\begin{pmatrix} u_{00} & u_{01} & u_{02} \\ 0 & u_{11} & u_{12} \\ 0 & 0 & u_{22} \end{pmatrix}$

Lower-triangular: $\begin{pmatrix} l_{00} & 0 & 0 \\ l_{10} & l_{11} & 0 \\ l_{20} & l_{21} & l_{22} \end{pmatrix}$

Gaussian elimination: example

UNIVERSITY OF CAMBRIDGE

Problem to solve:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 8 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

Add -4 times first row to the second:

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 7 & 8 & 8 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ -3 \\ 0 \end{pmatrix}$$

Add -7 times first row to the third:

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & -6 & -13 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ -3 \\ -7 \end{pmatrix}$$

Add -2 times second row to the third:

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ -3 \\ -1 \end{pmatrix}$$

We now have upper-triangular form.

Complexity is $O(n^3)$.

Forward/back substitution

We can now use back substitution to find the unknown values.

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ -3 \\ -1 \end{pmatrix}$$

1. The bottom row immediately gives $x_2 = 1$.
2. Substituting this into the second row, $-3x_1 - 6 = -3$, so $x_1 = -1$.
3. Finally, the first row gives $x_0 - 2 + 3 = 1$, so $x_0 = 0$.

Complexity is $O(n^2)$.

Forward substitution works in the same way, but for lower-triangular matrices instead of upper-triangular ones.

Gaussian elimination: pivoting

The very first step is to multiply the top line by $-\frac{A_{0,1}}{A_{0,0}}$.

What if the **pivot element** $A_{0,0}$ is zero or very small/big?

Dividing by a small pivot can blow up the result, and cause overflow in computations.

Solution: Rows can be freely interchanged without altering the equations.

Hence we are free to choose which remaining row to use for each outer loop step, so always choose the row with the largest leading value.

Selecting the best next row is **partial row pivoting**.

Various other quick processes can also be used before we start: scaling rows so all have similar magnitudes, or permuting columns. Column permutation requires we also permute the rows in $\mathbf{x}$ to preserve the result.

Permuting both rows and columns (**total pivoting**) can lead to more accurate solutions in computations, but the search complexity is undesirable.

LU factorisation

UNIVERSITY OF CAMBRIDGE

A popular factorization for solving linear systems, inverting matrices or computing determinants by a computer is to decompose into lower- and upper-triangular matrices $A = LU$, then

1. find $\mathbf{y}$ from $L\mathbf{y} = \mathbf{b}$ using forward substitution with the triangular form L,
2. find $\mathbf{x}$ from $U\mathbf{x} = \mathbf{y}$ using back substitution.

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

LU factorisation

Find L and U such that $A = LU$.

Dolittle algorithm:

Do a normal Gaussian elimination on A to get U (ignoring any r.h.s. to hand), but keeping track of the steps you would make on a r.h.s. in an extra matrix. The extra matrix turns out to be L.

More precisely, starting with $L = I$ and $U = A$:

For each row $\mathbf{u}_i$, with i from 1 to $m-1$ inclusive:

For each element $u_{i,j}$, with j from 0 to $i-1$ inclusive:

Set $u_{i,j}$ to zero by setting $\mathbf{u}_i = \mathbf{u}_i - \frac{u_{i,j}}{u_{j,j}}\mathbf{u}_j$

Set $l_{i,j} = \frac{u_{i,j}}{u_{j,j}}$

Complexity: $O(n^3)$. A minor overhead on top of Gaussian elimination.

LU factorisation: example

Recall the steps we took in the Gaussian elimination example. This time, maintain L and U so that they always multiply to give the original matrix.

$$
\begin{aligned}
A &= LU \\
\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 8 \end{pmatrix} &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 8 \end{pmatrix} \\
\text{[Add -4 times first row to the second]} &= \begin{pmatrix} 1 & 0 & 0 \\ \mathbf{4} & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ \mathbf{0} & -3 & -6 \\ 7 & 8 & 8 \end{pmatrix} \\
\text{[Add -7 times first row to the third]} &= \begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ \mathbf{7} & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ \mathbf{0} & -6 & -13 \end{pmatrix} \\
\text{[Add -2 times second row to the third]} &= \begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & \mathbf{2} & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & \mathbf{0} & -1 \end{pmatrix}
\end{aligned}
$$

LU factorisation

UNIVERSITY OF CAMBRIDGE

Unlike Gaussian elimination, $\mathbf{b}$ is not considered for decomposition. This is highly useful as the $A = LU$ can be reused to solve other linear systems for different vectors $\mathbf{b}$. Solving for each $\mathbf{b}$ requires $O(n^2)$ operations, which becomes faster than Gaussian elimination when multiple linear systems need to be solved for different $\mathbf{b}$.

A matrix implementation is as follows: Denote columns of L by the vectors $\mathbf{l}_k$ and the rows of U by the row vectors $\mathbf{u}_k^T$.

1. Set $A_0 = A$.
2. For all $k = 1, .., n$ set $\mathbf{u}_k^T$ to the kth row of A_{k-1} and $\mathbf{l}_k$ to the kth column of A_{k-1} scaled so that $L_{k,k} = 1$ (i.e. divide by the kth element on the diagonal of A_{k-1}).
3. Calculate $A_k = A_{k-1} - \mathbf{l}_k \mathbf{u}_k^T$

Vector operations. In practice it's useful to operate on vectors rather than multiple loops on scalars, as modern CPUs have vector processing units (e.g. AVX for Intel) that implement vector operations using fewer CPU cycles.

Problems? What if $A_{1,1}$ (or $u_{j,j}$) is zero? This will give a non-existing factorisation although one may exist (even when A is singular).

Solution? As with Gaussian elimination, we can permute rows. This corresponds to:

$$PA = LU$$

where P is a permutation matrix. This is called an LUP factorisation. It can be shown that any square matrix has an LUP decomposition for an appropriate choice of P, and that the decomposition is numerically stable. This is highly useful in practice.

Is that still agnostic to $\mathbf{b}$?

LU decomposition

UNIVERSITY OF CAMBRIDGE

- Any square matrix admits an LUP decomposition.

- If A is non-singular (invertible), then it admits an LU decomposition iff all leading principal minors are non-zero.

- If A is singular with rank $k < n$, then it admits an LU decomposition if (not iff) its k leading principal minors are non-zero.

- L is always non-singular (diagonal is all 1s). U will have A's rank.

LU decomposition

UNIVERSITY OF CAMBRIDGE

Besides solving $A\mathbf{x} = \mathbf{b}$, LU factorisation can be used to:

- Determine whether A has an inverse:

  A^{-1} exists iff all diagonal elements of L and U are non-zero.

- Calculate the inverse of A:

  $A^{-1} = U^{-1}L^{-1}$, with the inverse of triangular matrices being easy to compute.

- Calculate the determinant of A:

$$\det(A) = \det(L)\det(U) = \prod_{k=0}^{n-1} L_{k,k} \prod_{k=0}^{n-1} U_{k,k} = \prod_{k=0}^{n-1} U_{k,k}$$

Cholesky factorisation

UNIVERSITY OF CAMBRIDGE

If A is **symmetric** and **positive definite**, we can optimise LU factorisation by setting $U = L^T$. We only need to compute half the terms.

Positive definite: A is positive definite if the scalar $\mathbf{v}^T A\mathbf{v} > 0$ for all $\mathbf{v}$. We'll see later that this also means all eigenvalues of A are positive. Intuitively, this places restrictions on how far vectors can be rotated by matrix A.

We also have positive semi-definite (≥ 0), negative definite (< 0), negative semi-definite (≤ 0).

Luckily, these properties commonly arise in real problems, e.g. covariance matrices, physics, and any AA^T where A contains only real numbers.

Cholesky factorisation

UNIVERSITY OF CAMBRIDGE

We can find the elements of L by simply expanding out LL^T.

$$
\begin{aligned}
LL^T &= \begin{pmatrix} L_{00} & 0 & 0 \\ L_{10} & L_{11} & 0 \\ L_{20} & L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} L_{00} & L_{10} & L_{20} \\ 0 & L_{11} & L_{21} \\ 0 & 0 & L_{22} \end{pmatrix} \\
&= \begin{pmatrix} L_{00}^2 & & (\text{symmetric}) \\ L_{10}L_{00} & L_{10}^2 + L_{11}^2 & \\ L_{20}L_{00} & L_{20}L_{10} + L_{21}L_{11} & L_{20}^2 + L_{21}^2 + L_{22}^2 \end{pmatrix} = A
\end{aligned}
$$

Complexity: $O(n^3)$, but twice as fast as LU factorisation, and no need for pivoting.

Cholesky factorisation

The recursive formulas to find the entries of L are:

$$L_{j,j} = \sqrt{A_{j,j} - \sum_{k=0}^{j-1} L_{j,k}^2}$$

$$L_{i,j} = \frac{1}{L_{j,j}} \left(A_{i,j} - \sum_{k=0}^{j-1} L_{i,k} L_{j,k} \right) \qquad \text{for } i > j$$

And we get:

$$L = \begin{pmatrix} \sqrt{A_{00}} & 0 & 0 \\ A_{10}/L_{00} & \sqrt{A_{11} - L_{10}^2} & 0 \\ A_{20}/L_{00} & \left(A_{21} - L_{20}L_{10}\right)/L_{11} & \sqrt{A_{22} - L_{20}^2 - L_{21}^2} \end{pmatrix}$$

Cholesky factorisation: example

$$\begin{pmatrix} 1 & 2 & -1 \\ 2 & 8 & -8 \\ -1 & -8 & 11 \end{pmatrix} = \begin{pmatrix} L_{00}^2 & & \text{(symmetric)} \\ L_{10}L_{00} & L_{10}^2 + L_{11}^2 & \\ L_{20}L_{00} & L_{20}L_{10} + L_{21}L_{11} & L_{20}^2 + L_{21}^2 + L_{22}^2 \end{pmatrix}$$

$$(L_{00} = \sqrt{1}) \quad : \quad \begin{pmatrix} 1 & & \text{(symmetric)} \\ L_{10} & L_{10}^2 + L_{11}^2 & \\ L_{20} & L_{20}L_{10} + L_{21}L_{11} & L_{20}^2 + L_{21}^2 + L_{22}^2 \end{pmatrix}$$

$$(L_{10} = 2, L_{20} = -1) \quad : \quad \begin{pmatrix} 1 & & \text{(symmetric)} \\ 2 & 4 + L_{11}^2 & \\ -1 & -2 + L_{21}L_{11} & 1 + L_{21}^2 + L_{22}^2 \end{pmatrix}$$

And so on. We finish with:

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 2 & 0 \\ -1 & -3 & 1 \end{pmatrix}$$

Cholesky factorisation

UNIVERSITY OF CAMBRIDGE

Computing inverses. We can use Cholesky factorisation to find A^{-1} even when A is not symmetric. We first find the inverse of AA^T, which is always symmetric and positive-definite (why?). Then we do:

$$A^T(AA^T)^{-1} = A^{-1}$$

Test for positive-definiteness. Cholesky factorisation is the widely used method for testing if a matrix is positive definite, by testing if its Cholesky decomposition exists.

Complex matrices. The matrix A needs to be self-adjoint to admit a Cholesky decomposition (self adjoint or Hermitian means A is its own conjugate transpose, $A = A^* = \overline{A^T}$).

Any problems?

Ill-conditionedness

As we did with error propagation, it's important to know how much a small perturbation at the input can affect the output (we don't want a small change in the input to make our output explode!). Such small perturbations can arise from measurement error, or truncation error, rounding error, etc. from previous calculations.

The condition number of a numerical problem measures that aspect for the worst case. We can think of it as the *maximum ratio of the relative change in the output to the relative change in the input, for all possible inputs*.

A problem is **well**-conditioned if it has a **low** condition number.

A problem is **ill**-conditioned if it has a **large** condition number.

The condition number is a property of the problem, regardless of the algorithm used to solve the problem.

Ill-conditionedness

UNIVERSITY OF CAMBRIDGE

Before we look at condition numbers for linear systems, let's analyze functions. As per the definition above we can say the condition number is when we consider a small perturbation ϵ at the input, i.e. ϵ/x is the relative error at the input:

$$K(x) = \lim_{\epsilon \to 0} \left| \frac{\frac{f(x+\epsilon) - f(x)}{f(x)}}{\frac{\epsilon}{x}} \right| = \left| \frac{x f'(x)}{f(x)} \right|$$

Exercise: Discuss $f(x) = 1/(1-x)$ for x close to 1.

Ill-conditionedness

UNIVERSITY OF CAMBRIDGE

For linear systems where we look to solve $A\mathbf{x} = \mathbf{b}$ accurately, the condition number tells us how much $\mathbf{x}$ will change with small perturbations in $\mathbf{b}$.

The condition number of matrix A is the maximum ratio of the relative change in $\mathbf{x}$ to the relative change in $\mathbf{b}$, for all $\mathbf{b}$.

Let ϵ be a (vector) perturbation in $\mathbf{b}$, hence the relative change in $\mathbf{b}$ is $\|\epsilon\|/\|\mathbf{b}\| = \|\epsilon\|/\|A\mathbf{x}\|$. Assuming A is invertible, then $\mathbf{x} = A^{-1}\mathbf{b}$ and hence the relative change in $\mathbf{x}$ is $\|A^{-1}\epsilon\|/\|\mathbf{x}\|$. Thus, the condition number is:

$$K(A) = \max_{\mathbf{x},\epsilon\neq 0} \frac{\|A^{-1}\epsilon\|}{\|\mathbf{x}\|} \Big/ \frac{\|\epsilon\|}{\|A\mathbf{x}\|} = \max_{\mathbf{e}\neq 0} \frac{\|A^{-1}\epsilon\|}{\|\epsilon\|} \times \max_{\mathbf{x}\neq 0} \frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|}$$

The condition number can also be defined in terms of the matrix' eigenvalues (which we'll see later in the course) which provides a more sensible way to compute the condition number of a matrix.

Ill-conditionedness

Consider solving

$$\begin{aligned} x + 3y &= 17 \\ 2x - y &= 6 \end{aligned} \qquad \text{i.e.} \qquad \begin{pmatrix} 1 & 3 \\ 2 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 17 \\ 6 \end{pmatrix}$$

Multiply first equation (or matrix row) by 2 and subtract giving

$$0x + 7y = 34 - 6$$

Hence $y = 4$ and (so) $x = 5$. Geometrically, this just means finding where the two lines given by $x + 3y = 17$ and $2x - y = 6$ intersect. In this case things are all nice because the first line has slope -3, and the second line slope $1/2$ and so they are nearly at right angles to each other.

Ill-conditionedness

Remember, in general, that if

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} p \\ q \end{pmatrix}$$

Then

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} \begin{pmatrix} p \\ q \end{pmatrix} = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix} \begin{pmatrix} p \\ q \end{pmatrix}$$

Problems if $ad - bc$ is small (not necessarily in absolute terms; it's sufficient to be small relative to $a^2 + b^2 + c^2 + d^2$). The lines then are *nearly parallel*!

Ill-conditionedness

UNIVERSITY OF CAMBRIDGE

Consider the harmless-looking

$$\begin{pmatrix} 1.7711 & 1.0946 \\ 0.6765 & 0.4181 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} p \\ q \end{pmatrix}$$

Solving we get

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 41810000 & -109460000 \\ -67650000 & 177110000 \end{pmatrix} \begin{pmatrix} p \\ q \end{pmatrix}$$

Big numbers from nowhere! Small absolute errors in p or q will be greatly amplified in x and y.

Consider this geometrically: we are projecting 2-D to nearly 1-D: so getting back will be tricky.

Ill-conditionedness

e.g. Matlab given a singular matrix finds (due to rounding error) a spurious inverse (but at least it's professional enough to note this):

```
A = [ 16     3     2    13
       5    10    11     8
       9     6     7    12
       4    15    14     1 ];
>> inv(A)

Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 9.796086e-18.
ans = 1.0e+15 *
              0.1251    0.3753   -0.3753   -0.1251
             -0.3753   -1.1259    1.1259    0.3753
              0.3753    1.1259   -1.1259   -0.3753
             -0.1251   -0.3753    0.3753    0.1251
```

Note the 10^{15}!!

Ill-conditionedness in matrix factorizations: Cholesky

UNIVERSITY OF CAMBRIDGE

If the matrix A is very ill-conditioned then the classic Cholesky decomposition is problematic, as it requires taking square roots: some of the numbers under the square roots can become negative in computations due to (rounding) error accumulation, though they are always positive theoretically. This can be addressed by adding a diagonal matrix D to the factorisation:

$$
\begin{aligned}
A &= LDL^T \\
&= \begin{pmatrix} 1 & 0 & 0 \\ L_{10} & 1 & 0 \\ L_{20} & L_{21} & 1 \end{pmatrix} \begin{pmatrix} D_0 & 0 & 0 \\ 0 & D_1 & 0 \\ 0 & 0 & D_2 \end{pmatrix} \begin{pmatrix} 1 & L_{10} & L_{20} \\ 0 & 1 & L_{21} \\ 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} D_1 & & \text{(symmetric)} \\ L_{10}D_0 & L_{10}^2 D_0 + D_1 & \\ L_{20}D_0 & L_{20}L_{10}D_0 + L_{21}D_1 & L_{20}^2 D_0 + L_{21}^2 D_1 + D_2 \end{pmatrix}
\end{aligned}
$$

Ill-conditionedness in matrix factorizations: Cholesky

The recursive formulas to find the entries of D and L are:

$$\begin{aligned} D_j &= A_{jj} - \sum_{k=0}^{j-1} L_{jk}^2 D_k \\ L_{ij} &= \frac{1}{D_j}\left(A_{ij} - \sum_{k=0}^{j-1} L_{ik} L_{jk} D_k\right) \quad \text{for } i > j \end{aligned}$$

Note the lack of square roots! This gives:

$$L = \begin{pmatrix} 1 & 0 & 0 \\ \frac{A_{10}}{D_0} & 1 & 0 \\ \frac{A_{20}}{D_0} & \frac{A_{21} - L_{20} L_{10} D_0}{D_1} & 1 \end{pmatrix} \quad D = \begin{pmatrix} A_{00} & 0 & 0 \\ 0 & A_{11} - L_{10}^2 D_0 & 0 \\ 0 & 0 & A_{22} - L_{20}^2 D_0 - L_{21}^2 D_1 \end{pmatrix}$$

Same number of values as the classic Cholesky decomposition (just spread over 3 matrices instead of 2), so this has the same $O(n^3)$ cost.

QR factorisation

UNIVERSITY OF CAMBRIDGE

Another factorisation $A = QR$, but we will create a matrix with a different useful property. We will construct Q to be **orthogonal**. R is upper-triangular.

Why are orthogonal matrices useful? Recall:

- Their transpose is also their inverse: $QQ^T = I$
- They preserve the 2-norm of vectors (energy): $\|Q\mathbf{v}\|^2 = \|\mathbf{v}\|^2$
- They preserve scalar products: $Q\mathbf{v} \cdot Q\mathbf{w} = \mathbf{v} \cdot \mathbf{w}$

This means that transforming a matrix or vector with Q does not drastically change the size of its elements. The process is *numerically stable*.

Rotations and reflections are examples of transformations which can be represented using orthogonal matrices.

QR factorisation

UNIVERSITY OF CAMBRIDGE

Once we have this factorisation, we can solve:

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ QR\mathbf{x} &= \mathbf{b} \\ \text{Solve } Q\mathbf{y} &= \mathbf{b}, \text{ where } \mathbf{y} = R\mathbf{x} \\ Q^T Q\mathbf{y} &= Q^T\mathbf{b} \\ \mathbf{y} &= Q^T\mathbf{b} \\ \text{Solve } R\mathbf{x} &= \mathbf{y} \text{ using back substitution} \end{aligned}$$

Also works when A (and R) is **tall**, i.e. $m \times n$ with $m \geq n$.

How do we "build" orthogonal vectors?

To obtain Q we have to build a set of n orthogonal vectors starting from the vectors a_k of A, regarding $A = (\mathbf{a}_0, ..., \mathbf{a}_{n-1})$. The following procedure can also be used to generate random orthogonal matrices, which is in fact another popular usage of the QR factorisation (randomize A then use QR).

QR factorisation: Gram-Schmidt algorithm

UNIVERSITY OF CAMBRIDGE

This is one method for performing QR factorisation, with complexity $O(m^3)$.

We proceed in a similar way to Gaussian elimination, but instead of subtracting to create zeros, we subtract to make the columns orthogonal to each other.

The algorithm processes one column of A $(m \times n)$, Q $(m \times m)$ and R $(m \times n)$ at a time, so it can be easier to think of the matrices in terms of their column vectors:

$$A = (\mathbf{a}_0, ..., \mathbf{a}_{n-1}), Q = (\mathbf{q}_0, ..., \mathbf{q}_{m-1}), R = (\mathbf{r}_0, ..., \mathbf{r}_{n-1})$$

QR factorisation: Gram-Schmidt algorithm

UNIVERSITY OF CAMBRIDGE

Normalise the first column to have unit magnitude:

$$\mathbf{q}_0 = \frac{\mathbf{a}_0}{\|\mathbf{a}_0\|}, \quad \mathbf{r}_0 = (\|\mathbf{a}_0\|, 0, 0, ..., 0)$$

From $\mathbf{a}_1$ build a vector $\mathbf{w_1}$ orthogonal to $\mathbf{q}_0$ (and therefore to $\mathbf{a}_0$), i.e. find the component of $\mathbf{a}_1$ that is orthogonal to $\mathbf{q}_0$:

$$\mathbf{w}_1 = \mathbf{a}_1 - (\mathbf{q}_0 \cdot \mathbf{a}_1)\mathbf{q}_0 \quad \text{subtract } \mathbf{a}_1\text{'s component along } \mathbf{q}_0$$

$$\mathbf{q}_1 = \frac{\mathbf{w}_1}{\|\mathbf{w}_1\|}, \quad \mathbf{r}_1 = (\mathbf{q}_0 \cdot \mathbf{a}_1, \|\mathbf{w}_1\|, 0, ..., 0)$$

Continue through the remaining columns:

$$\mathbf{w}_k = \mathbf{a}_k - \sum_{i=0}^{k-1} (\mathbf{q}_i \cdot \mathbf{a}_k)\mathbf{q}_i$$

$$\mathbf{q}_k = \frac{\mathbf{w}_k}{\|\mathbf{w}_k\|}, \quad \mathbf{r}_k = (\mathbf{q}_0 \cdot \mathbf{a}_k, \mathbf{q}_1 \cdot \mathbf{a}_k, ..., \mathbf{q}_{k-1} \cdot \mathbf{a}_k, \|\mathbf{w}_k\|, 0, ..., 0)$$

QR factorisation: Gram-Schmidt algorithm example

UNIVERSITY OF CAMBRIDGE

Let's work through the following example:

$$A = \begin{pmatrix} 1 & 1 & 1 \\ -1 & 0 & 1 \\ 1 & 1 & 2 \end{pmatrix} = QR = \begin{pmatrix} q_{00} & q_{01} & q_{02} \\ q_{10} & q_{11} & q_{12} \\ q_{20} & q_{21} & q_{22} \end{pmatrix} \begin{pmatrix} r_{00} & r_{01} & r_{02} \\ 0 & r_{11} & r_{12} \\ 0 & 0 & r_{22} \end{pmatrix}$$

First column: normalise $\mathbf{a}_0$:

$$\|\mathbf{a}_0\| = \sqrt{1^2 + (-1)^2 + 1^2} = \sqrt{3}$$

$$\begin{pmatrix} 1 & 1 & 1 \\ -1 & 0 & 1 \\ 1 & 1 & 2 \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{3}} & q_{01} & q_{02} \\ -\frac{1}{\sqrt{3}} & q_{11} & q_{12} \\ \frac{1}{\sqrt{3}} & q_{21} & q_{22} \end{pmatrix} \begin{pmatrix} \sqrt{3} & r_{01} & r_{02} \\ 0 & r_{11} & r_{12} \\ 0 & 0 & r_{22} \end{pmatrix}$$

QR factorisation: Gram-Schmidt algorithm example

UNIVERSITY OF CAMBRIDGE

Second column:

$$
\begin{aligned}
\mathbf{q}_0 \cdot \mathbf{a}_1 &= \frac{1}{\sqrt{3}} \cdot 1 + -\frac{1}{\sqrt{3}} \cdot 0 + \frac{1}{\sqrt{3}} \cdot 1 = \frac{2}{\sqrt{3}} \\
\mathbf{w}_1 &= (1,0,1) - \frac{2}{\sqrt{3}} \left(\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}} \right) = \left(\frac{1}{3}, \frac{2}{3}, \frac{1}{3} \right) \\
\|\mathbf{w}_1\| &= \sqrt{\left(\frac{1}{3}\right)^2 + \left(\frac{2}{3}\right)^2 + \left(\frac{1}{3}\right)^2} = \frac{\sqrt{6}}{3}
\end{aligned}
$$

$$
\begin{pmatrix} 1 & 1 & 1 \\ -1 & 0 & 1 \\ 1 & 1 & 2 \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & q_{02} \\ -\frac{1}{\sqrt{3}} & \frac{2}{\sqrt{6}} & q_{12} \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & q_{22} \end{pmatrix} \begin{pmatrix} \sqrt{3} & \frac{2}{\sqrt{3}} & r_{02} \\ 0 & \frac{\sqrt{6}}{3} & r_{12} \\ 0 & 0 & r_{22} \end{pmatrix}
$$

QR factorisation: Gram-Schmidt algorithm example

UNIVERSITY OF CAMBRIDGE

Third column:

$$
\begin{aligned}
\mathbf{q}_0 \cdot \mathbf{a}_2 &= \frac{1}{\sqrt{3}} \cdot 1 + -\frac{1}{\sqrt{3}} \cdot 1 + \frac{1}{\sqrt{3}} \cdot 2 = \frac{2}{\sqrt{3}} \\
\mathbf{q}_1 \cdot \mathbf{a}_2 &= \frac{1}{\sqrt{6}} \cdot 1 + \frac{2}{\sqrt{6}} \cdot 1 + \frac{1}{\sqrt{6}} \cdot 2 = \frac{5}{\sqrt{6}} \\
\mathbf{w}_2 &= (1,1,2) - \frac{2}{\sqrt{3}}\left(\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right) - \frac{5}{\sqrt{6}}\left(\frac{1}{\sqrt{6}}, \frac{2}{\sqrt{6}}\frac{1}{\sqrt{6}}\right) = \left(-\frac{1}{2}, 0, \frac{1}{2}\right) \\
\|\mathbf{w}_2\| &= \sqrt{\left(-\frac{1}{2}\right)^2 + 0^2 + \left(\frac{1}{2}\right)^2} = \frac{\sqrt{2}}{2}
\end{aligned}
$$

$$
\begin{pmatrix} 1 & 1 & 1 \\ -1 & 0 & 1 \\ 1 & 1 & 2 \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & -\frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{3}} & \frac{2}{\sqrt{6}} & 0 \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} \sqrt{3} & \frac{2}{\sqrt{3}} & \frac{2}{\sqrt{3}} \\ 0 & \frac{\sqrt{6}}{3} & \frac{5}{\sqrt{6}} \\ 0 & 0 & \frac{\sqrt{2}}{2} \end{pmatrix}
$$

Finished!

QR factorisation: Gram-Schmidt algorithm

- If we ever find a $\mathbf{w}_k$ which is $\mathbf{0}$, we do not add it to Q – it means that $\mathbf{a}_k$ is not orthogonal to the previous components.
- If the above happens, Q will not be full when all columns of A have been processed. In that case, fill the remaining columns in any way that makes the matrix orthogonal (how?). The corresponding rows of R will be all zero.

Error amplification: Small errors in the dot products of the QR steps (summing up many numbers) can grow (rapidly) and hinder the orthogonality of Q; the theoretical zeros of $Q^T Q = I$ can get quite a bit larger than zero.

Solution: Orthogonality is preserved if we decompose Q into a product of two orthogonal matrices (factoriception?). The curious can lookup the techniques of *Givens rotations* or *Householder reflections*.

Comparison of techniques for solving linear systems

UNIVERSITY OF CAMBRIDGE

- **Matrix is triangular?**
  Use in that form.
- **Matrix rows OR columns can be permuted to be triangular?**
  Use the permutation.
- **Matrix rows AND columns can be permuted to be triangular?**
  Requires an expensive search unless the matrix is sparse.
- **Matrix is symmetric and positive definite?**
  Use Cholesky factorisation.
- **None of the above, but the matrix is square?**
  - one r.h.s.: use Gaussian Elimination.
  - multiple r.h.s.: use LU factorisation.

Comparison of techniques for solving linear systems

UNIVERSITY OF CAMBRIDGE

- **Interested in computing the determinant or inverse of a matrix?** (Or just checking whether an inverse exists?)
  Use LU factorisation.
- **Want to check if a matrix is positive-definite?**
  Use Cholesky factorisation (and see if it fails).
- **Matrix is tall (overspecified)?**
  Use QR factorisation.
- **Matrix is fat (underspecified)?**
  Select the best solution under some metric function (coming up soon).

Least Squares

Least Squares (LS) is a method to approximate solutions to overdetermined systems (more equations than unknowns), i.e. situations where we typically acquire more measurements than there are variables in the system. It is most used in *data fitting*, *regression* or *prediction* by minimizing the sum of squared residuals (why squares?). There are linear and non-linear LS methods.

Here we'll start from simple linear examples, and without much mathematical formalism, then formalize the method later. Let's take a first easy example.

You have a spring and want to estimate its force constant, knowing that the deformation is proportional to the applied force

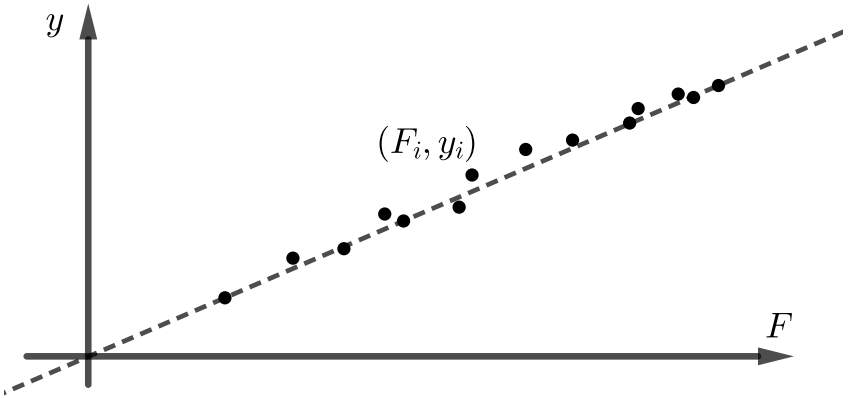
$$y = kF.$$

The above is our system *model* or *model function*. You now take n measurements where you apply known force values F_i and measure the deformation y_i corresponding to each, obtaining

$$y_i = kF_i + \epsilon_i,$$

where ϵ_i are statistical errors uncorrelated with F_i, with mean 0, constant variance and not autocorrelated (more on this later), that the measurement device and/or process have introduced. In LS lingo, F is called the independent/predictor/control variable and y is called the dependent/predicted variable (these are just a few of ridiculously many used namings: `https://en.wikipedia.org/wiki/Dependent_and_independent_variables#Statistics_synonyms`).

We want to fit a line that *best fits* this dataset, that is we want to find an estimate $\hat{k}$ of k (the slope of the line) that gives a line of best fit:



Fitting a line in the LS sense means that we want to find $\hat{k}$ as the "best" estimate of k, such that the sum of squared differences between each

measurement y_i and kF_i for all k is minimum. These differences are called *residuals* for some value of k (see also the Note below):

$$r_i = y_i - kF_i.$$

In other words our estimate $\hat{k}$ will be the value of k which minimizes

$$S(k) = \sum_{i=1}^{n} r_i = \sum_{i=1}^{n} (y_i - kF_i)^2,$$

i.e.

$$\hat{k} = \underset{k}{\operatorname{argmin}}\, S(k).$$

To find $\hat{k}$ we differentiate $S(k)$ to get

$$\frac{dS}{dk} = \frac{d}{dk} \sum_i (y_i - kF_i)^2 = -2 \sum_i F_i (y_i - kF_i)$$

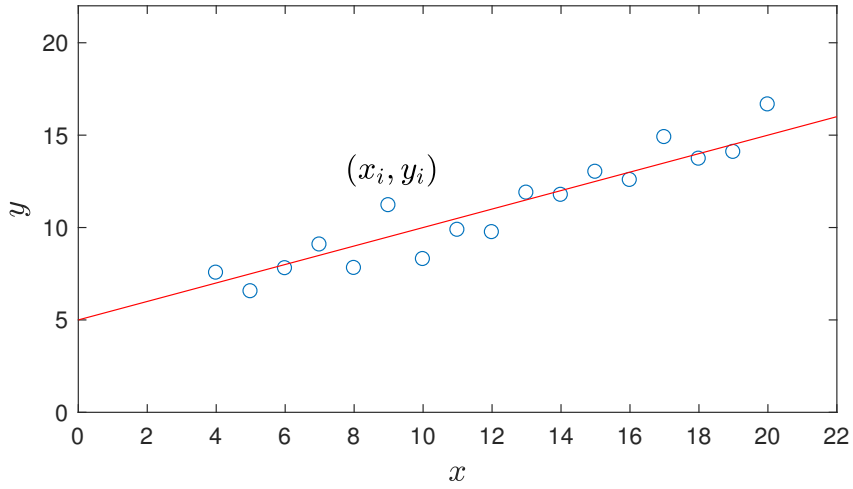
and equating to 0 we get

$$k = \frac{\sum_i F_i y_i}{\sum_i F_i^2}.$$

Compare this with the poor man's approach (PMA) where we compute $\hat{k}$ as the sample mean of all ratios y_i/F_i, i.e. $\hat{k} = \frac{1}{n}\sum_i y_i/F_i$, after randomizing ϵ_i.

Note or errors vs residuals: The errors ϵ_i and the residuals r_i are not the same (!), though they can be easily confused in literature, mostly due to notation used in the equations they appear. The errors ϵ_i are statistical errors, i.e. the differences between the measurements and the true (unobservable!) values. The residuals r_i are the differences between the measurements and the predicted (fitted) values from our assumed model. We can define residuals r_i for any value of k, not just $\hat{k}$ (the one that gives the best fit in the LS sense), which is why the notation above did not say $r_i = y_i - \hat{k}F_i$, but instead $r_i = y_i - kF_i$ with k to be found later via minimization.

What if we had an experiment where our points were shifted up by some constant?



We now switch to the popular notation in LS literature, where β represent model parameters to be found (previously k), and x represent control variables.

In this case we would assume that the system's model is

$$y = \beta_0 + \beta_1 x,$$

i.e. β_0 is the shift (intercept) and β_1 is the slope. The measurements are

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i,$$

and we need to estimate both β_0 and β_1 by finding $\hat{\beta}_0$ and $\hat{\beta}_1$ that minimize the sum of squared residuals

$$r_i = y_i - (\beta_0 + \beta_1 x_i).$$

This time the function S is of two variables

$$S(\beta_0, \beta_1) = \sum_i r_i^2 = \sum_i (y_i - \beta_0 - \beta_1 x_i)^2$$

and we use partial derivatives in turn wrt β_0 and β_1 in order to find $\hat{\beta}_0$ and $\hat{\beta}_1$ as the solutions to:

$$\frac{\partial S}{\partial \beta_1} = 0 \qquad \text{and} \qquad \frac{\partial S}{\partial \beta_0} = 0$$

which, after a bit of algebra give:

$$\hat{\beta}_1 = \frac{\sum_i x_i y_i - \frac{1}{n}\sum_i x_i \sum_i y_i}{\sum_i x_i^2 - \frac{1}{n}(\sum_i x_i)^2} \qquad \text{and} \qquad \hat{\beta}_0 = \frac{1}{n}\sum_i y_i - \hat{\beta}_1 \frac{1}{n}\sum_i x_i.$$

The above is known as *simple linear regression* and is by far the most widely known and used form of fitting/regression in a plethora of applications. One can recognize that the numerator and denominator of $\hat{\beta}_1$ are the sample covariance and sample variance, while the terms in β_0 are the sample means.

What about a poor man's approach (PMA) in this case? We can extract $\hat{\beta}_0$ and $\hat{\beta}_1$ as follows. Compute the sample means $x_m = \frac{1}{n}\sum_i x_i$ and $y_m = \frac{1}{n}\sum_i y_i$. Shift all the data points down and left by y_m and x_m respectively, and then compute the slope $\hat{\beta}_1$ as the mean of ratios $\hat{\beta}_1 = \frac{1}{n}\sum_i \frac{y_i - y_m}{x_i - x_m}$ and $\hat{\beta}_0$ directly as $\hat{\beta}_0 = y_m - \hat{\beta}_1 x_m$ (the latter is the same as the least squares' $\hat{\beta}_0$). Randomize a few examples and observe what happens to the PMA fitted line. What can you say about the accuracy and why?

Can we fit any polynomial, or other functions?

UNIVERSITY OF CAMBRIDGE

The short answer is yes (but we should be careful about high order polynomials; we'll see why later). All the above models are called linear LS because they are linear in the parameters β_j (and **not** in x or y). For example, we can apply a quadratic fit model:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2$$

and it would be a linear model, because it is linear in β_j. We would define the residuals r_i and the residual function S in the same way, $S(\beta_0, \beta_1, \beta_2) = \sum_i (y_i - \beta_0 - \beta_1 x_i - \beta_2 x_i^2)^2$ and take three partial derivatives to find $\hat{\beta}_0, \hat{\beta}_1, \hat{\beta}_2$ as the solutions to:

$$\frac{\partial S}{\partial \beta_0} = 0 \quad \text{and} \quad \frac{\partial S}{\partial \beta_1} = 0 \quad \text{and} \quad \frac{\partial S}{\partial \beta_2} = 0$$

all of which are simple linear equations, hence easy to solve (algebra gets more tedious on paper).

Always linear equations: As long as our model is linear in β_j then we will always get linear equations from the partial derivatives, owing to the choice of "squares" as the metric for our minimization problem (this is sometimes used as an answer to "why squares?"). The "coefficient" of β_j can be any function $\phi_j(x)$, not just polynomial powers, and the model would be a linear LS model, i.e.

$$y = \sum_{j=1}^{p} \beta_j \phi_j(x),$$

is a general form for a linear LS model with a single independent/input variable (x) and p model parameters (β_j) that we need to estimate. For example, $y = \beta_0 \cos x + \beta_1 e^x$ is yet another example of a linear LS model, with measurements (observed values) $y_i = \beta_0 \arctan x_i + \beta_1 e^{x_i} + \epsilon_i$, where ϵ_i are statistical errors, uncorrelated with x_i, with mean 0. In short, if

$$\frac{\partial S}{\partial \beta_j} = C$$

where C is a constant wrt all β_j then we have a linear LS model.

Reducing to line fit. When we have only two parameters to estimate β_0 and β_1 and one of them is just a shift (intercept), i.e. the model is $y = \beta_0 + \beta_1\phi(x)$, where $\phi(x)$ is an arbitrary function, then we can see this is a line fit by writing $z = \phi(x)$ and $z_i = \phi(x_i)$, reusing the results obtained previously for fitting a line, then substituting at the end. LS is completely agnostic to the 'coefficient' of β_1.

If ϕ is bijective, and hence invertible, then we can also 'replot' the points x_i on the x-axis as $\phi^{-1}(x_i)$ to turn the curve $\beta_0 + \beta_1\phi(x)$ into a straight line.

Multiple independent variables x

UNIVERSITY OF CAMBRIDGE

The underlying model of a system may consist of multiple independent variables, or you may want to want to predict data taking into account multiple variables (e.g. car sales depending on gas mileage, top speed, engine power, price, etc). We shall first give a similar algebraic form as until now, but then formalise it more succinctly using vectors and matrices. In this case, our linear model becomes

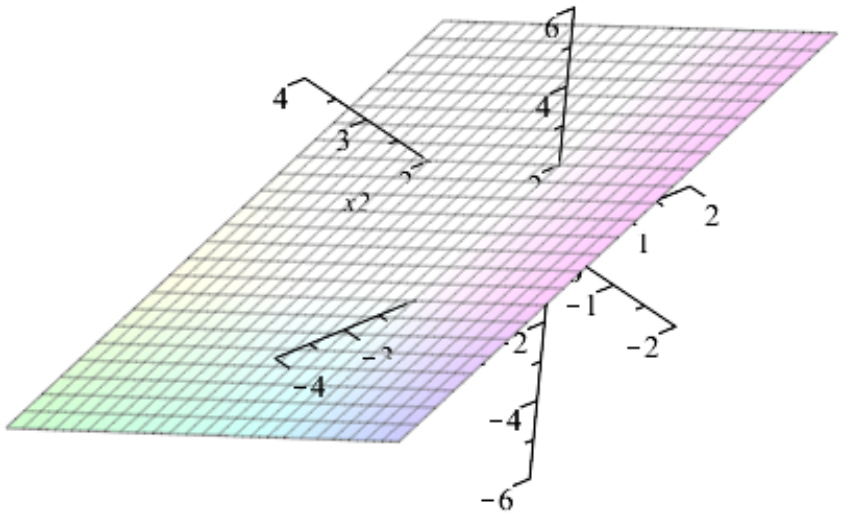
$$y = \beta_0 x_0 + \cdots + \beta_p x_p \tag{3}$$

and the measurements become

$$y_i = \beta_0 x_{0i} + \cdots + \beta_p x_{pi} + \epsilon_i. \tag{4}$$

For example, the following would give a plane in 3D space:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$



$$y = 2 + x_1 - x_2$$

and the measurements $y_i = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \epsilon_i$ would give points above & below (hopefully close enough) to this plane depending on the statistical errors ϵ_i. This plane is the true model, i.e. unknown to us, which we try to estimate via linear LS, i.e. we try to fit a plane $\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2$ (instead of a curve in 2D) through the points y_i in 3D space.

For any linear problem with p independent variables, we perform the same steps as before, extracting the residuals $r_i = y_i - \sum_j \beta_j x_{ji}$ and minimizing the

objective function

$$S(\beta) = \sum_i r_i^2 = \sum_i (y_i - \sum_j \beta_j x_{ji})^2$$

by taking all partial derivatives

$$\frac{\partial S}{\partial \beta_k} = 0 \quad \text{for all } k = 1, \ldots, p$$

to find the estimates $\hat{\beta}_k$. Being linear, the above p equations are straightforward

$$-2 \sum_i x_{ki}(y_i - \sum_j \beta_j x_{ji}) = 0 \tag{5}$$

which give

$$\hat{\beta}_k = \frac{\sum_i x_{ki}(y_i - \sum_{j \neq k} \hat{\beta}_j x_{ji})}{\sum_i x_{ki}^2}. \tag{6}$$

Functions of x_j

UNIVERSITY OF CAMBRIDGE

The previous case of a single independent variable is a special case of multiple independent variables, by taking some of the variables as functions of previous variables. For example, the single variable quadratic model $y = \beta_0 + \beta_1 x + \beta_2 x^2$ is obtained by making $x_0 = 1$ (i.e. kept constant and equal to 1 throughout all measurements) and $x_2 = x_1^2$.

Moreover, similar to the single variable case, any of the variables can be functions, i.e. some $\phi_j(x_j)$ instead of x_j, and the model would still be linear, since it's linearity in β_j that matters. For example

$$y = \beta_0 \arctan x_0 + \beta_1 x_1^3 + \beta_2 e^{x_2}$$

is a linear model with 3 independent variables. By making the change of variable $z_0 = \arctan x_0, z_1 = x_1^3, z_2 = e^{x_2}$ we reuse the same result for the estimates $\hat{\beta}_j$ and substitute back at the end.

Thus a fully general form for the linear LS model is:

$$y = \sum_{j=1}^{p} \beta_j \phi_j(x_j),$$

though we shall stick to $y = \sum_{j=1}^{p} \beta_j x_j$ for simplicity of notation.

Matrix formulation of linear LS

The n equations (4) with $i = 1, \ldots, n$ can be written in matrix form directly as

$$\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_{11} & \cdots & x_{p1} \\ \vdots & \ddots & \vdots \\ x_{1n} & \cdots & x_{pn} \end{pmatrix} \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_p \end{pmatrix} + \begin{pmatrix} \epsilon_1 \\ \vdots \\ \epsilon_n \end{pmatrix} \tag{7}$$

or more simply:

$$\underline{y} = X\underline{\beta} + \underline{\epsilon}. \tag{8}$$

The above is a linear system of equations, with more equations (n) than unknowns (p). As we know, it does not have a unique solution, the point of LS is to estimate one that gets us close to the measurements y_i.

From here on we shall only work with vectors and matrices, and shall drop the underscores, that is:

$$y = X\beta + \epsilon. \tag{9}$$

Similarly, the residuals become

$$r = y - X\beta \tag{10}$$

and the objective function to minimize becomes

$$S(\beta) = \sum_i r_i^2 = \|r\|^2 = \|y - X\beta\|^2 \tag{11}$$

and we equate its gradient to 0 (differentiation steps below are left as an exercise):

$$\begin{aligned}
\nabla S(\beta) &= \nabla (y - X\beta)^T (y - X\beta) \\
&= \nabla (y^T - \beta^T X^T)(y - X\beta) \\
&= \nabla (y^T y - y^T X\beta - \beta^T X^T y + \beta^T X^T X\beta) \\
&= -X^T y - X^T y + 2X^T X\beta \\
&= -2X^T (y - X\beta) \\
&= 0
\end{aligned} \tag{12}$$

which gives the famous closed form of the linear LS estimator:

$$\hat{\beta} = (X^T X)^{-1} X^T y. \tag{13}$$

Why does this make sense?

Let's see how this makes sense, and in the process also confirm Gauss's choice to minimize the sum of squared residuals.

Take as an example a linear LS model with $p = 2$ independent variables and $n = 3$ measurements, i.e. the vectors y, ϵ have 3 elements, and the matrix X would be 3-by-2. The following figure shows this case, though we can extrapolate it to any n, p (planes become hyperplanes).

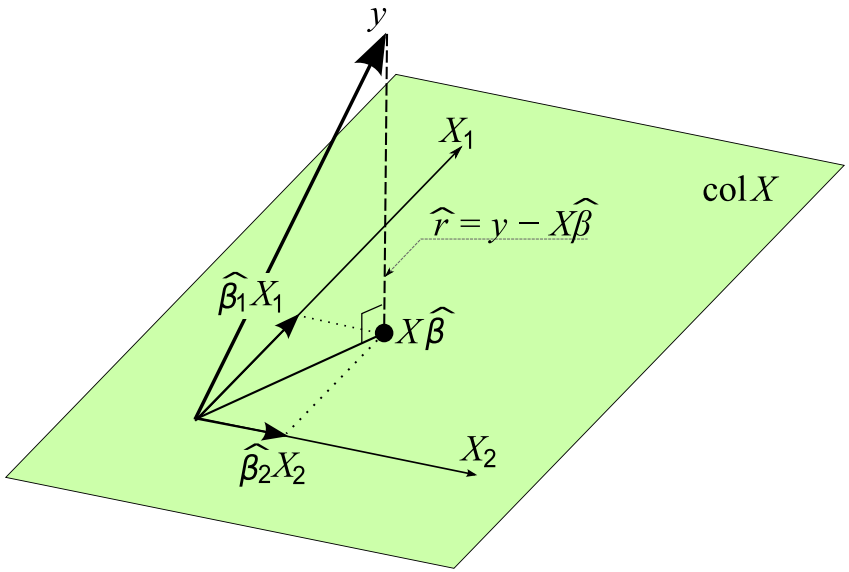


image adapted from Wikipedia

Let us write the matrix X as $X = [X_1\ X_2]$, i.e. the vectors X_1 and X_2 are the the 2 columns of X. These 2 vectors span a subspace of R^3, shown as the green plane in the figure. The vector y of measurements does not live in this subspace (if it did, then it would necessarily be a linear combination of X_1 and X_2, i.e. the system $y = X\beta$ would have a unique solution). We have free choice of the coefficients β_1 and β_2 to make the vector $X_1\beta_1 + X_2\beta_2 = X\beta$

land on any point in this subspace. Since we live in this subspace, we will never be able to get to y (to solve the linear system exactly) so the best we can do is to get as close as possible to y, while staying in the subpace. That point must be the orthogonal projection of y onto this subspace, since that gives the shortest distance to our subspace, i.e. the vector $y - X\hat{\beta}$. Being orthogonal to our subspace, it is orthogonal to all vectors in this subspace, in particular X_1 and X_2 (which spanned the subspace to begin with), i.e.

$$X_1^T(y - X\hat{\beta}) = 0 \qquad \text{and} \qquad X_1^T(y - X\hat{\beta}) = 0$$

or simply

$$X^T(y - X\hat{\beta}) = 0$$

This is exactly the same condition that we derived before by minimizing the objective function $S(\beta)$ which shows that the sum of squared residuals is justified as a choice for an objective function to minimize.

The matrix X

Obviously, X^TX needs to be invertible. It is a square symmetric matrix (called

the Gramian matrix of X). If it is not invertible (singular) then it means some of its columns are a linear combination of the other columns. We can show (*) that if this is the case, then it implies that the columns of X itself (which is the matrix of our independent variables' values) are also linearly dependent. That is, not all of our assumed independent variables are actually independent, i.e. some of our variables are in fact redundant, and we need to estimate fewer parameters β_j than we thought, in which case we need to redo our model and recompute the above with reduced X, y, β.

(*) If the columns of $X^T X \in \mathbb{R}^{p \times p}$ are linearly dependent then there is a vector $v \in \mathbb{R}^p$ such that $X^T X v = 0$. Left multiplying by v^T we get $v^T X^T X v = ||Xv||^2 = 0$, hence $Xv = 0$, hence the columns of X itself are also linearly dependent.

Cholesky to compute the estimate. Attempt a Cholesky decomposition LL^T of $X^T X$, which we know is symmetric. Cholesky requires $X^T X$ to also be positive definite. If it fails then $X^T X$ is singular, so we know we have redundant variables used in the model (X has redundant columns). If it succeeds then we can invert LL^T efficiently as $(L^T)^{-1} L^{-1}$, to obtain our linear LS estimate $\hat{\beta}$.

Example 1

Find an approximate solution to $X\beta = y$ where:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 2 & 1 \end{bmatrix} \quad \text{and} \quad y = \begin{bmatrix} 6 \\ 0 \\ 0 \end{bmatrix}.$$

We have

$$X^T X = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 5 & 3 \\ 3 & 3 \end{bmatrix}$$

and

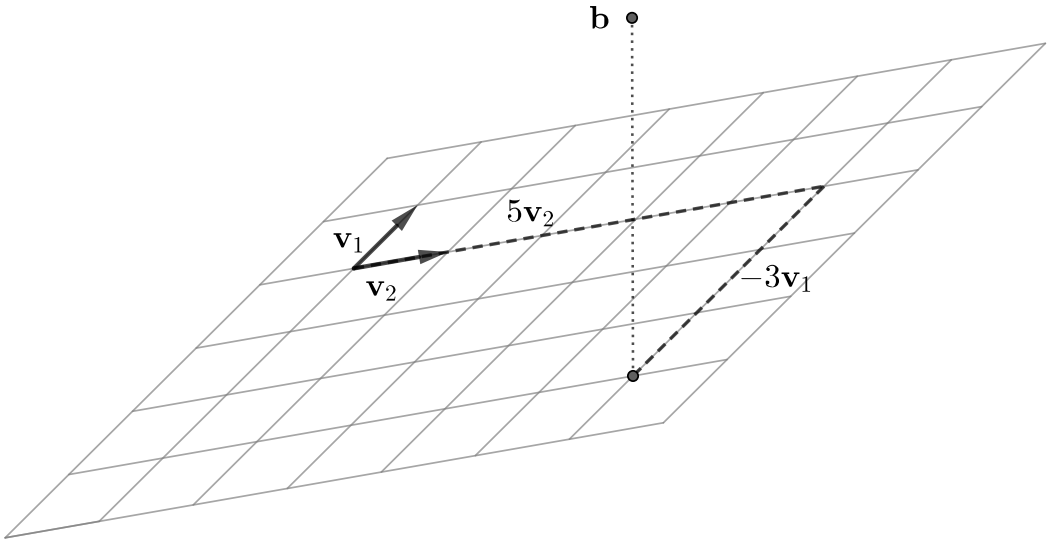
$$X^T y = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 6 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 6 \end{bmatrix}.$$

Gaussian/ Direct/ Cholesky/ LU/ QR/ etc to solve

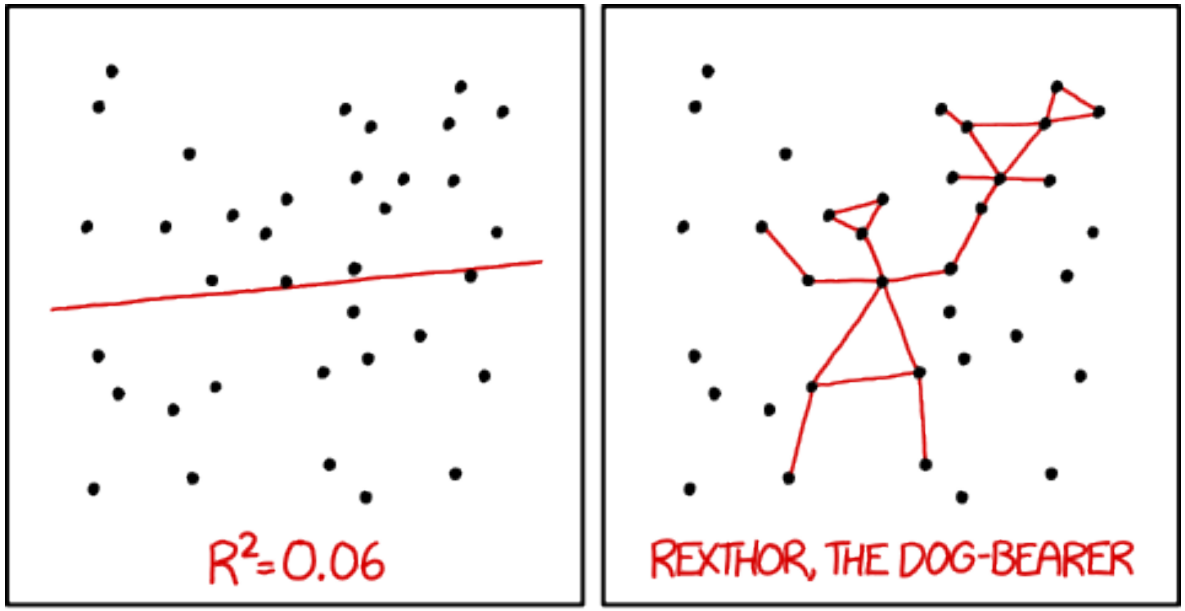
$$\begin{bmatrix} 5 & 3 \\ 3 & 3 \end{bmatrix} \beta = \begin{bmatrix} 0 \\ 6 \end{bmatrix}$$

and obtain

$$\hat{\beta} = \begin{bmatrix} -3 \\ 5 \end{bmatrix}.$$



Goodness of fit: R^2



I DON'T TRUST LINEAR REGRESSIONS WHEN IT'S HARDER TO GUESS THE DIRECTION OF THE CORRELATION FROM THE SCATTER PLOT THAN TO FIND NEW CONSTELLATIONS ON IT.

A measure for goodness of fit is the so called *coefficient of determination* denoted by R^2 (the square is misleading, as this quantity can get negative depending on the model), defined as:

$$R^2 = 1 - \frac{\sum_i r_i^2}{\sum_i (y_i - y_m)^2} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - y_m)^2},$$

where $y_m = \frac{1}{n}\sum_i y_i$ is the sample mean, and $\hat{y} = X\hat{\beta}$ is our fitted model. The fraction above is called the *fraction of unexplained variance*, because it is the ratio between the variance of the model's residuals, called the *unexplained variance*, and the *total variance* of the data itself.

Interpretation. A value of $R^2 = 0.7$ can be interpreted as saying that 70% of the variance in the dependent (output) variable y can be explained by the independent (input) variables x_j. The other 30% can be attributed to unknown, confounding variables (variables that influence both the independent and dependent variables) or inherent variability.

Alternative definition: Sometimes R^2 is defined differently, as:

$$R^2 = \frac{\sum_i (\hat{y}_i - y_m)^2}{\sum_i (y_i - y_m)^2},$$

to mean the fraction of *explained variance* instead. Note that the two definitions are not necessarily equal. They are equal when the two variances add up to the total variance, which reduces to $y^T \bar{y} = \hat{y}^T \bar{y}$, where $\bar{y} = [y_m, \ldots, y_m]$.

Inflation of R^2 and adjusted R^2. If we add a new independent variable $\beta_{p+1} x_{p+1}$ to our model, then the minimum sum of squared residuals usually decreases, it never increases! (**) This is an artificial increase in accuracy of our fitted model and is an artefact of the LS method. It is a problem because it means we can add a totally random and unrelated variable (e.g. the first letter of the driver of the car in the car sales estimation experiment) and make our model appear to have a better fit, which would be misleading.

To compensate for this, we can define a so called *Adjusted* R^2, which scales the

unexplained variance according to the number of variables in the model:

$$R^2_{\text{adj}} = 1 - \frac{\frac{1}{n-p-1}\sum_i r_i^2}{\frac{1}{n-1}\sum_i (y_i - y_m)^2} = 1 - (1-R^2)\frac{n-1}{n-p-1}.$$

The above inflation can also be understood at a high level as follows: adding a new independent variable means we add an extra degree of freedom to our model (e.g. from a line fit to a plane fit). This gives us more headroom for optimization (e.g. we can tilt the plane along the line) to obtain a lower minimum of fit. In general, a constrained problem gives worse results than an unconstrained problem, but is generally more efficient to compute.

(**) Similar to extracting the minimum value from a list of numbers: adding a new number to the list cannot increase the previous minimum but may lower it. Here, we obtain the new fitted model by minimizing the objective function $S_{\text{new}}(\beta_1, \ldots, \beta_p, \beta_{p+1})$. If we force $\beta_{p+1} = 0$ when minimizing then we obtain the minimum of the original objective function $S(\beta_1, \ldots, \beta_p)$. Thus, when β_{p+1} is unconstrained, the minimum of the new objective function S_{new} is necessarily lower or maximum equal, i.e. the new model can never be a worse fit.

Note 1: R^2 shows correlation, not causality. It does not mean that x_i necesasrily cause y. For example, there is correlation between icecream sales and shark attacks (true story) ... obviously, other variables not taken into account are causing the latter or have more relevant correlation (e.g. sunny weather).

Note 2: R^2 can be negative if an intercept term is not included ($x_{0i} \neq 1$), because it can cause a bad fit, enough to cause the sum of squared residuals to be greater than the data's total variance. In general, R^2 gets negative if the model is a sufficiently bad fit. The definition of R^2 in statistics as *"the ratio between the explained variance and total variance"* for the simple LS model in fact holds only when an intercept term is included.

Least Squares Applications

UNIVERSITY OF CAMBRIDGE

- `http://setosa.io/ev/ordinary-least-squares-regression/`

- **Interpolation/extrapolation, outliers, overfitting, low/high variance, ill-conditioned X (e.g. Vandermonde matrix for polynomial fit)**

- **Recursive prediction**

- ~~Homoscedasticity and weighted LLS~~ (no time)

- ~~Gradient descent, global minimum~~ (no time, though intuitive)

- ~~Nonlinear LS~~ (no time)

- Underdetermined systems, regularization (moved to exercises)

Eigenvalues & Eigenvectors

Recall Linear Transformations

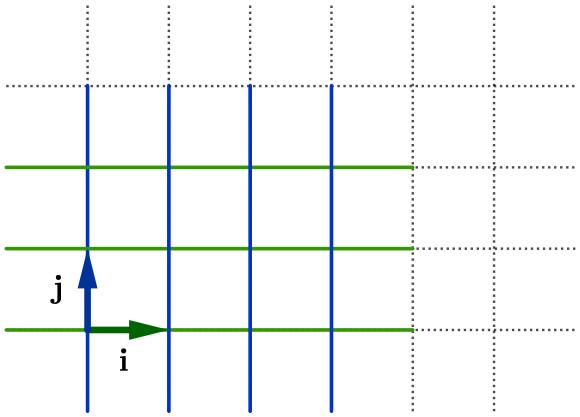
UNIVERSITY OF CAMBRIDGE

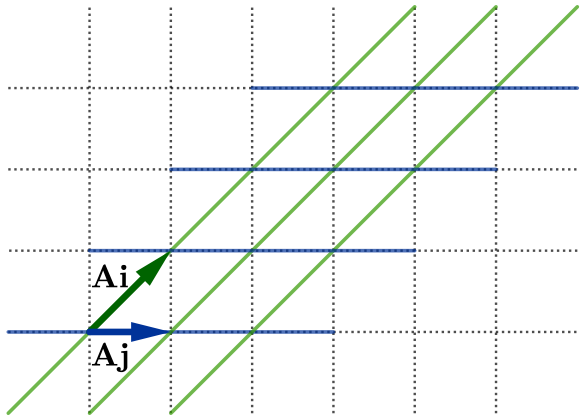
We start with a short summary of *some* of the linear algebra properties covered in the Maths course. Consider a simple vector $\mathbf{v} \in \mathbb{R}^2$ and a matrix $A \in \mathbb{R}^{2\times 2}$:

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix},$$

Think of A as a linear transformation intrinsic to A, i.e. it *skews* any vector v in some way (scale, rotate, reflect, flatten, etc). In other words, think of the product $\mathbf{Av}$ as of applying a linear transformation associated with $\mathbf{A}$ to the vector $\mathbf{v}$. To visualise said transformation, we look at what $\mathbf{A}$ does to the canonical basis vectors $\mathbf{i}$ and $\mathbf{j}$ of the two-dimensional Euclidean space:

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} \overset{\mathbf{A}}{\longmapsto} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 0 \\ 1 \end{bmatrix} \overset{\mathbf{A}}{\longmapsto} \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$





$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} \overset{\mathbf{A}}{\longmapsto} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix} \overset{\mathbf{A}}{\longmapsto} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Now, observe an arbitrary vector $\mathbf{v}$,

$$\mathbf{v} = \begin{bmatrix} 3 \\ 2 \end{bmatrix},$$

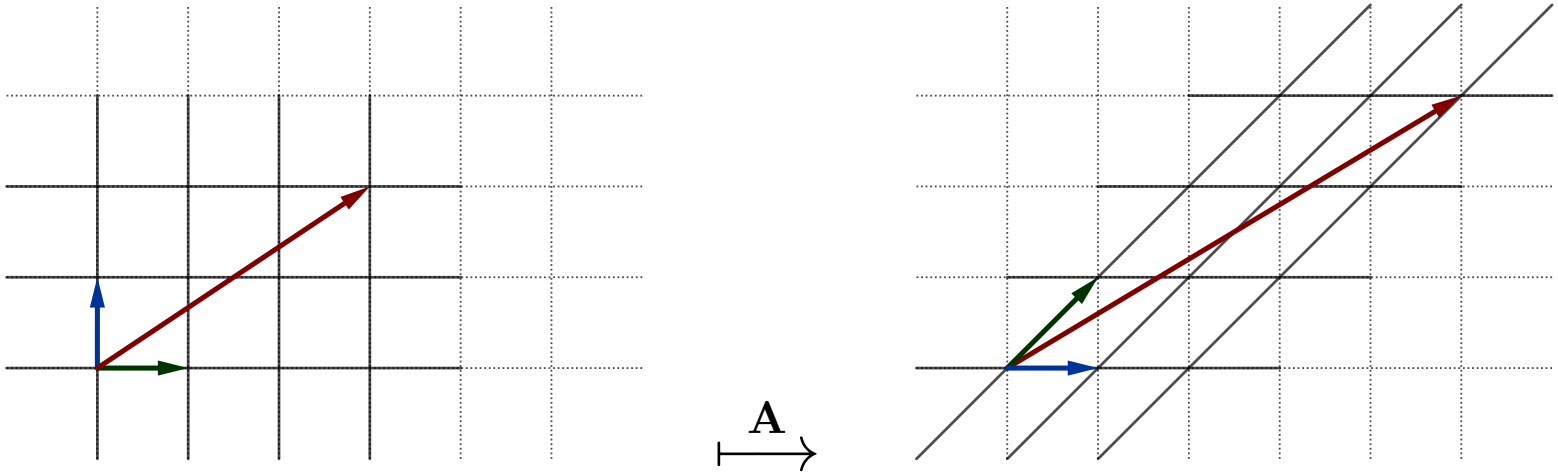
We can represents this vector using the basis vectors $\mathbf{i}$ and $\mathbf{j}$,

$$\mathbf{v} = 3\begin{bmatrix} 1 \\ 0 \end{bmatrix} + 2\begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Since $\mathbf{A}$ represents a linear transformation,

$$\mathbf{A}\mathbf{v} = \mathbf{A}\left(3\begin{bmatrix} 1 \\ 0 \end{bmatrix} + 2\begin{bmatrix} 0 \\ 1 \end{bmatrix}\right) = 3\mathbf{A}\begin{bmatrix} 1 \\ 0 \end{bmatrix} + 2\mathbf{A}\begin{bmatrix} 0 \\ 1 \end{bmatrix} = 3\begin{bmatrix} 1 \\ 1 \end{bmatrix} + 2\begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

It is now rather easy to visualise what happens with vector $\mathbf{v}$ under the transformation $\mathbf{A}$.



Notice that although $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 5 \\ 3 \end{bmatrix}$, i.e. $\mathbf{A} : \begin{bmatrix} 3 \\ 2 \end{bmatrix} \longmapsto \begin{bmatrix} 5 \\ 3 \end{bmatrix}$, the coordinates of vector $\mathbf{v}$ with respect to $(\mathbf{i}, \mathbf{j})$ and $(\mathbf{Ai}, \mathbf{Aj})$ did not change; this depends on A, though.

General case

In general, let

$$\mathbf{A} = [\mathbf{a}_1 \ \mathbf{a}_2 \ \cdots \ \mathbf{a}_n]$$

be an $n \times n$ matrix where $\mathbf{a_k} \in \mathbb{R}^n$ are column vectors and let $\mathbf{v} \in \mathbb{R}^n$. We can think of the product $\mathbf{Av}$ as of applying the linear transformation associated

with $\mathbf{A}$ to the vector $\mathbf{v}$. This linear transformation maps the canonical (Euclidean space) basis vectors $\mathbf{i}_k$ to vectors $\mathbf{a}_k$ respectively.

For any linear transformation $\mathbf{A}$ and any scalars α, β we have,

$$\mathbf{A}(\alpha\mathbf{x} + \beta\mathbf{y}) = \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{A}\mathbf{y}.$$

Given a vector $\mathbf{v} = (v_1, v_2, \ldots, v_n)^T \in \mathbb{R}^n$ we have

$$\begin{aligned}\mathbf{A}\mathbf{v} = \mathbf{A}(v_1\mathbf{i}_1 + v_2\mathbf{i}_2 + \cdots + v_n\mathbf{i}_n) &= \mathbf{A}v_1\mathbf{i}_1 + \mathbf{A}v_2\mathbf{i}_2 + \cdots + \mathbf{A}v_n\mathbf{i}_n \\ &= v_1\mathbf{A}\mathbf{i}_1 + v_2\mathbf{A}\mathbf{i}_2 + \cdots + v_n\mathbf{A}\mathbf{i}_n \\ &= v_1\mathbf{a}_1 + v_2\mathbf{a}_2 + \cdots + v_n\mathbf{a}_n.\end{aligned}$$

In other words, the coordinates (elements) of the vector v with respect to the vectors $\mathbf{i}_k$ are the same as the coordinates (elements) with respect to the vectors $\mathbf{a}_k$.

Note: While the vectors $\mathbf{i}_k$ form a basis, the vectors $\mathbf{a}_i$ do not necessarily form a basis unless they are linearly independent, i.e. the matrix A is non-singular. If

A is singular, then the vectors $[\mathbf{a}_k]$ will be linearly dependent (and vice-versa) and the representation of $\mathbf{v}$ will not be unique (i.e. there exists $\mathbf{v}_1 \neq \mathbf{v}_2$ such that $\mathbf{A}\mathbf{v_1} = \mathbf{A}\mathbf{v_2}$). In other words, the columns $\mathbf{a}_k$ of A will span the same space as $\mathbf{i}_k$ if and only if A is non-singular (plus the rest of the properties seen in the Maths course).

Eigenvectors and eigenvalues

Are there any vectors $\mathbf{v} \neq 0$ such that $\mathbf{A}\mathbf{v}$ and $\mathbf{v}$ have the same direction? If yes, $\mathbf{v}$ are called **eigenvectors** of $\mathbf{A}$ (the linear transformation $\mathbf{A}$ has preferred directions). That is, we ask if there are any scalars $\lambda \neq 0$ such that

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}. \tag{14}$$

This is equivalent to

$$\mathbf{A}\mathbf{v} = (\lambda\mathbf{I})\mathbf{v}$$
$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = 0$$
$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0.$$

The solutions λ_i to the above equation are called **eigenvalues** of $\mathbf{A}$. The above gives a polynomial, called the *characteristic polynomial*, and hence it can be written as $P(x) = a\prod_i(x - \lambda_i)$ for some scalar a. Using the coefficients of the characteristic polynomial to find its roots can be a very ill-conditioned problem (lookup Wilkinson's polynomial) even if the original problem is well-conditioned (coefficients can get extremely large, even though the roots themselves are all far apart) due to round-off errors, which are increasingly problematic as n grows.

For $n \geq 5$ there are no closed form solutions to the characteristic polynomial (Abel-Ruffini theorem) and hence eigenvalues must be computed using **iterative methods**. However, even for $n = 3$ the exact formula is impractical.

To obtain the **eigenvectors** or $\mathbf{A}$, we solve the linear systems $(\mathbf{A} - \lambda_i \mathbf{I})\mathbf{v} = \mathbf{0}$ for each λ_i (using any of the methods we learned so far, e.g. factorization), and obtain the solutions $\mathbf{v}_i$. These are then the **eigenvectors** or $\mathbf{A}$.

Note 1: Eigenvalues and eigenvectors can be complex valued even if $\mathbf{A}$ is real valued. Alternatively, we may also say that $\mathbf{A}$ does not have eigenvalues or eigenvectors.

Note 2: Depending on A some eigenvalues and eigenvectors may not be distinct.

Let's take the previous example. Applying equation (14) gives:

$$\det\left(\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} - \lambda \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) = \lambda^2 - \lambda - 1 = 0$$

with its two famous solutions:

$$\lambda_{1,2} = \frac{1 \pm \sqrt{5}}{2}.$$

These are the eigenvalues. To obtain the eigenvectors, we solve $(\mathbf{A} - \lambda_i \mathbf{I})\mathbf{v} = \mathbf{0}$ for each λ_i:

$$\left(\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} - \lambda_i \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) \mathbf{v} = \begin{bmatrix} 1-\lambda_i & 1 \\ 1 & -\lambda_i \end{bmatrix} \mathbf{v} = \mathbf{0}, \quad i \in \{1, 2\}.$$

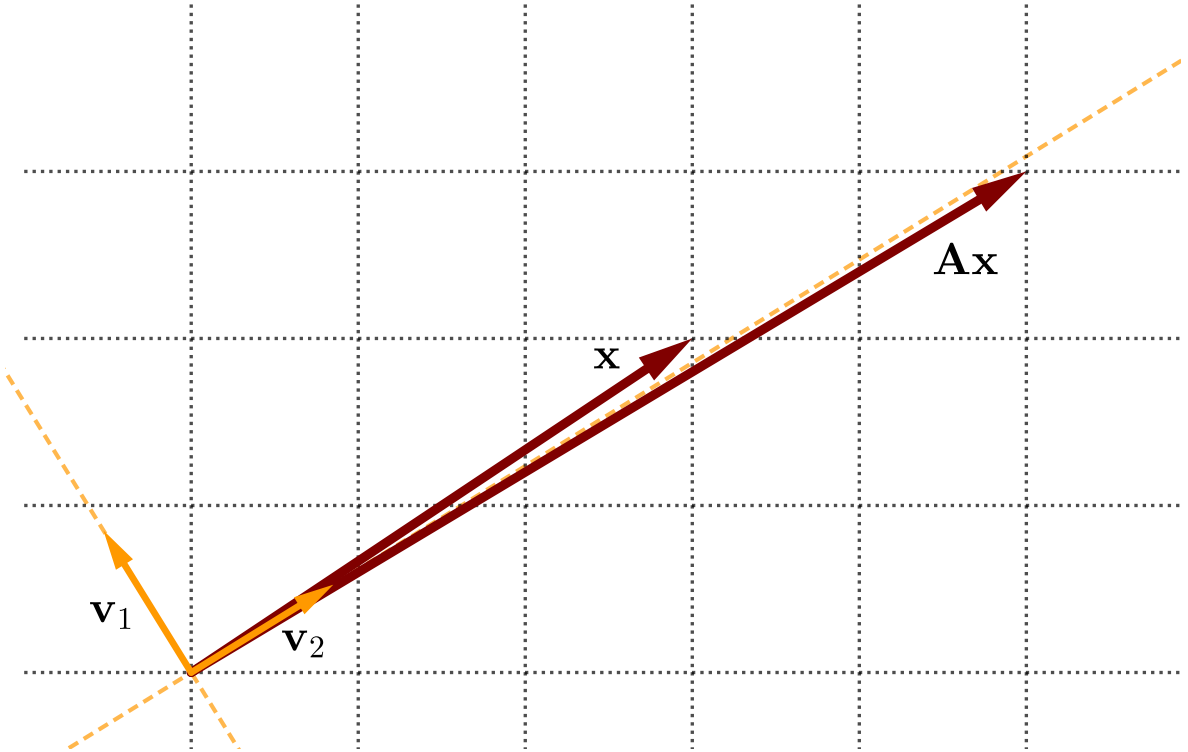
Solving these two linear systems gives

$$\mathbf{v}_1 = p \begin{bmatrix} \varphi_1 \\ 1 \end{bmatrix} \quad \text{and} \quad \mathbf{v}_2 = q \begin{bmatrix} \varphi_2 \\ 1 \end{bmatrix} \quad \text{for any } p, q \in \mathbb{R} \setminus \{0\}.$$

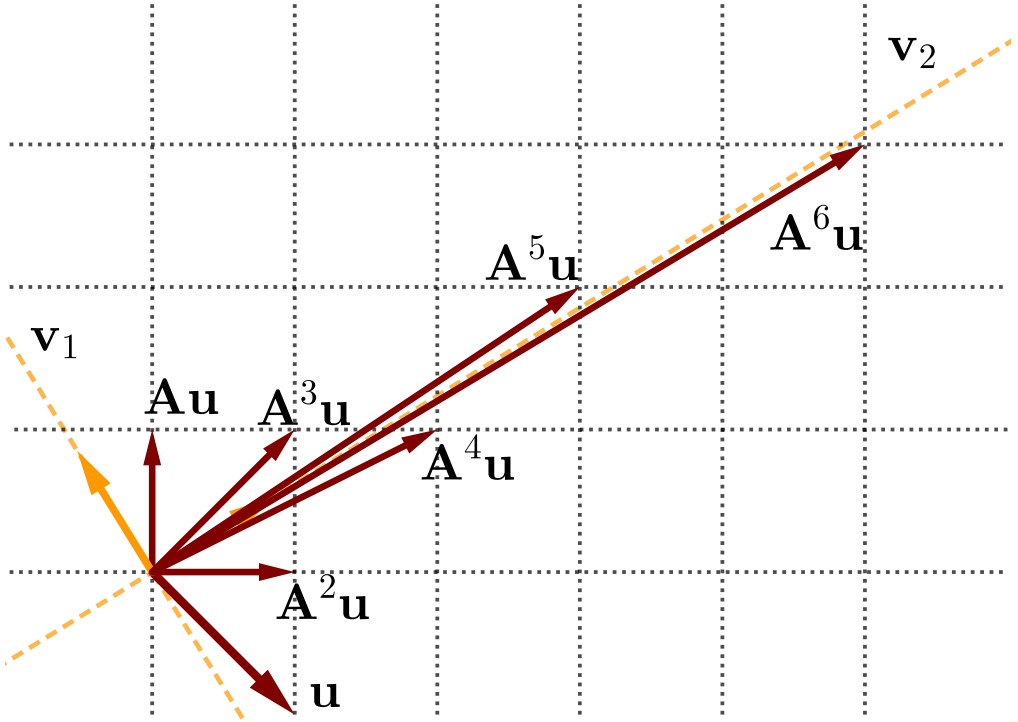
These are the **eigenvectors**. Notice that for any $\mathbf{A}$ and scalar c, we have $\mathbf{A}(c\mathbf{v}) = c\mathbf{A}\mathbf{v}$. In other words, if $\mathbf{v}$ is an eigenvector, so is $c\mathbf{v}$. We usually want the ones with unit length, which in the above example are

$$\mathbf{v}_i = \frac{\begin{bmatrix} \lambda_i \\ 1 \end{bmatrix}}{\left\| \begin{bmatrix} \lambda_i \\ 1 \end{bmatrix} \right\|}, \qquad i \in \{1, 2\}.$$

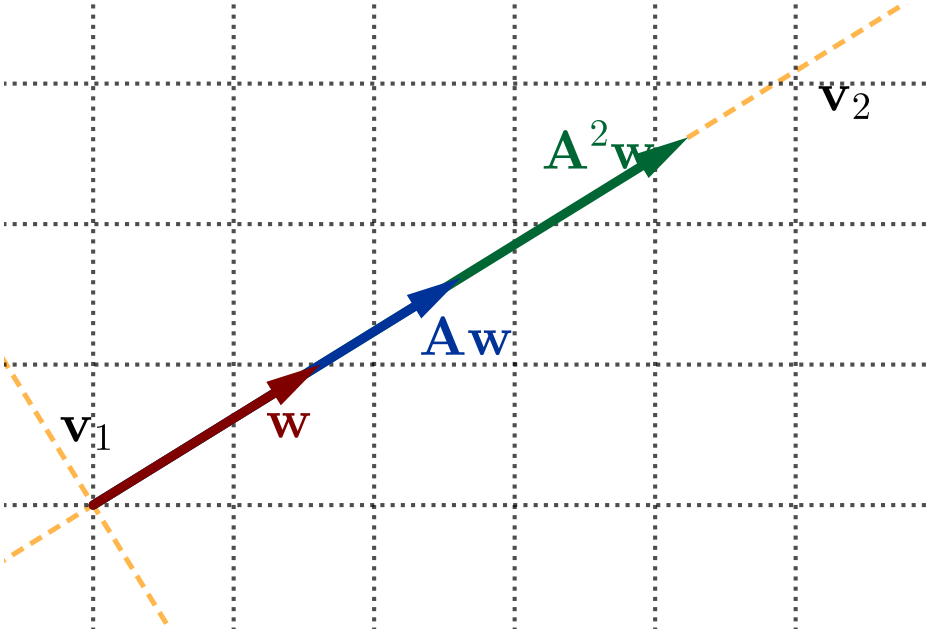
Example 1. Eigenvectors $\mathbf{v}_1$ and $\mathbf{v}_2$, and the previous vector $\mathbf{x} = [3, 2]^T$. Notice that the direction of $\mathbf{x}$ did not change much after applying $\mathbf{Ax}$, because it is close to to one of the eigenvectors (direction of $[3, 2]^T$ is the same as the direction of $[1.5, 1]^T$ which is close to $[\varphi, 1]^T \approx [1.618, 1]^T$).



Example 2. Let's observe the same transformation $\mathbf{A}$ applied successively to vector $\mathbf{u} = [1, -1]^T$, i.e. observe the vectors $\mathbf{Au}, \mathbf{A}^2\mathbf{u}, \mathbf{A}^3\mathbf{u}, \mathbf{A}^4\mathbf{u}, \ldots$. They get closer and closer to one of the eigenvectors. This is an iterative method for computing the largest eigenvalue (and its corresponding eigenvector(s)), called **power iteration**.



Exercices: Explain why does it alternate direction around $\mathbf{v}_2$. What would happen if $\mathbf{u}$ was collinear with one of the eigenvectors?



Example 3. If $\mathbf{w}$ was collinear with $\mathbf{v}_2$ then $\mathbf{w} = c\mathbf{v}_2$ for some scalar c, i.e. $\mathbf{w}$ is also an eigenvector. Applying $\mathbf{A}$ successively will preserve direction and scale it by λ_2 each time, i.e. a geometric progression with ratio λ_2

$$\frac{\|\mathbf{A}\mathbf{w}\|}{\|\mathbf{w}\|} = \frac{\|\mathbf{A}^2\mathbf{w}\|}{\|\mathbf{A}\mathbf{w}\|} = \cdots = \frac{\|\mathbf{A}^k\mathbf{w}\|}{\|\mathbf{A}^{k-1}\mathbf{w}\|} = \lambda_2.$$

A matrix $\mathbf{A}$ will scale vectors along the eigenvector direction with the scaling factor equal to the corresponding eigenvalue.

Computing eigenvalues and eigenvectors

UNIVERSITY OF CAMBRIDGE

Power iteration, as seen in the above example is an iterative way to compute eigenvalues. It shows that computing eigenvectors first (before eigenvalues) is more tractable for computers. Start with a random vector b_0 and iterative via

$$b_{k+1} = \frac{Ab_k}{\|Ab_k\|}.$$

This assumes that A has an eigenvalue greater than all others, and that b_0 is not orthogonal on the eigenvector corresponding to said eigenvalue (i.e. has a non-zero component along that direction). In that case, the algorithm gives both the eigenvector $b = \lim_{k\to\infty} b_k$ and the eigenvalue $\lambda = \frac{b^T Ab}{b^T b}$. A variation is to multiply with $(A - \mu I)^{-1}$ instead of A, which will converge to an eigenvalue close to μ. This algorithm is slow (linear/1st order convergence), and depends on the initial choice.

The advent of the **QR algorithm** in 1961 was a dramatic change (there were no efficient and accurate methods until then). The QR algorithm was named

one of the 10 most influential algorithms of the 20th century. Using QR to find the eigenvectors and eigenvalues is an iterative process that is fast, stable and accurate. Start with $A_0 = A$ then for each $k \geq 0$ iterate via

$$A_k = Q_k R_k,$$
$$A_{k+1} = R_k Q_k,$$

where $Q_k R_k$ is the QR factorisation of A_k. Since $Q_k^T Q_k = I$, we have

$$A_{k+1} = Q_k^T Q_k R_k Q_k = Q_k^T A_k Q_k,$$

hence all A_k have the same eigenvalues. It can be shown that as this algorithm converges, A_k converge to an upper triangular matrix, hence the eigenvalues are simply its diagonal entries.

Exercise 1: Let A, B, P be n-by-n matrices, and P be invertible. Show that if $B = P^{-1}AP$ (we say A and B are 'similar') then B and A have the same eigenvalues.

Exercise 2: Show that the eigenvalues of a triangular matrix are its diagonal entries.

Orthogonality and eigenbases

UNIVERSITY OF CAMBRIDGE

In our particular example we also have

$$\mathbf{v}_1 \cdot \mathbf{v}_2 = \varphi_1 \varphi_2 + 1 = 0,$$

i.e. the eigenvectors are orthogonal. This holds when A is real and symmetric (the orthogonality theorem seen in the Maths course). If A is a real symmetric n-by-n matrix then there is a set of n eigenvectors of A that form an orthonormal basis of $\mathbb{R}^n$. The reverse is also true (it is in fact an *iff* statement). If the eigenvalues are all different (all have multiplicity 1) then any set of eigenvectors corresponding to the eigenvalues will be orthogonal.

For a general n-by-n matrix A, eigenvectors corresponding to different eigenvalues form a basis, called an *eigenbasis*, but not necessarily an orthogonal one.

Practice Question

Take the following matrix

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

Start computing its first few powers by hand: A^2, A^3, etc. What pattern do you see? Can you explain why this pattern shows up? Is there an efficient way to compute arbitrary powers of this matrix, A^n for any n? Given that two eigenvectors of this matrix are

$$\mathbf{v}_1 = \begin{bmatrix} 2 \\ 1+\sqrt{5} \end{bmatrix} \quad \mathbf{v}_2 = \begin{bmatrix} 2 \\ 1-\sqrt{5} \end{bmatrix}$$

see if you can figure out a way to compute A^n by first changing to an eigenbasis, compute the new representation of A^n in that basis, then converting back to our standard basis. What does this expression tell you?

Eigen* Applications

UNIVERSITY OF CAMBRIDGE

- `http://setosa.io/ev/eigenvectors-and-eigenvalues/`
- Google PageRank
- Principle Components Analysis (Trump & Brexit!), `http://setosa.io/ev/principal-component-analysis/`
- Image compression
- ~~Image feature extraction~~ (no time)

Floating point representation

Based on slides by David Greaves from 2018.

Review: signed and unsigned integers

UNIVERSITY OF CAMBRIDGE

An 8-bit value such as 10001011 can naturally be interpreted as either an unsigned number $(2^7 + 2^3 + 2^1 + 2^0 = 139)$ or as a signed number $(-2^7 + 2^3 + 2^1 + 2^0 = -117)$.

This places the decimal (binary!?!) point at the right-hand end. It could also be interpreted as a *fixed-point number* by imagining a decimal point elsewhere (e.g. in the middle) to get 1000.1011; this would have value $2^3 + 2^{-1} + 2^{-3} + 2^{-4} = 8\frac{11}{16} = 8.6875$.

(The above is an unsigned fixed-point value for illustration, normally we use signed fixed-point values.)

Representations of non-integer values

UNIVERSITY OF CAMBRIDGE

Non-integer values can be represented using **fixed point** or **floating point**.

A fixed point number has a fixed number of bits reserved for the integer part and the fractional part, regardless its size.

Since an n bit representation can only represent 2^n numbers, choosing where to place the decimal point is a trade-off between precision and range. Fixed point representations fall short when it comes to storing small numbers precisely.

Floating point introduction

UNIVERSITY OF CAMBRIDGE

All floating point numbers we are interested in can be written as

$$\pm m \times \beta^e.$$

We call m the *mantissa* (or significand), β the *base*, and e the *exponent*.

By allowing the exponent to change, we can place the radix point anywhere we like among the value's significant digits (even outside them), i.e. the radix point can *float*.

Floating point representation

UNIVERSITY OF CAMBRIDGE

Floating point representation follows scientific notation. *If* we allocate a fixed size of storage for every number then we need to

- fix a size of mantissa (sig.figs.)
- fix a size for exponent (exponent range)

We also need to agree what the number means, e.g. agreeing the base used by the exponent.

Standards

UNIVERSITY OF CAMBRIDGE

In the past, every manufacturer produced their own floating point hardware and floating point programs gave different answers. IEEE standardisation fixed this.

There are two different IEEE standards for floating-point computation.

IEEE 754 is a binary standard that requires $\beta = 2, p = 24$ (number of mantissa bits) for *single precision* and $p = 53$ for *double precision*. It also specifies the precise layout of bits in a single and double precision.

In 2008 it was augmented to include additional longer binary floating point formats and also decimal floating formats.

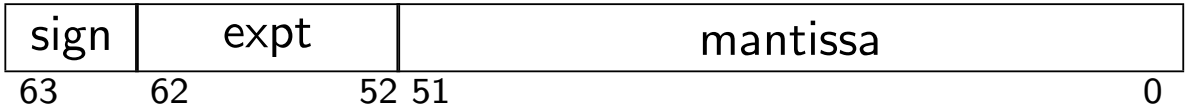
IEEE 854 is more general and allows binary and decimal representation without fixing the bit-level format.

IEEE 754 Floating Point Representation

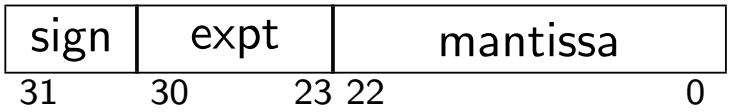
UNIVERSITY OF CAMBRIDGE

Here I give the version used on x86 (similar issues arise as in the ordering of bytes with an 32-bit integer).

Double precision: 64 bits (1+11+52), $\beta = 2, p = 53$



Single precision: 32 bits (1+8+23), $\beta = 2, p = 24$



Value represented is *typically*: $(s\ ?\ -1 : 1) * 1.mmmmmm * 2^{eeeee}$.

Note **hidden bit**: 24 (or 53) sig.bits, only 23 (or 52) stored!

$$(-1)^{sign}(1.b_{51}b_{50}...b_0)_2 \times 2^{e-1023} := (-1)^{sign}\left(1 + \sum_{i=1}^{52} b_{52-i}2^{-i}\right) \times 2^{e-1023}$$

Hidden bit and exponent representation

UNIVERSITY OF CAMBRIDGE

Advantage of base-2 ($\beta = 2$) exponent representation: all normalised numbers start with a '1', so no need to store it.

Like base 10 where normalised numbers start 1..9, in base 2 they start 1..1.

Can you see any difficulties this might cause?

Hidden bit and exponent representation

If all numbers start with a 1, how can we represent zero?

We need to cheat, and while we're at it, let's create representations for some other special values too.

Zero Exponent $= 0$, mantissa $= 0$

Infinity Exponent $=$ maximum (e.g. 11111111), mantissa $= 0$

Denormalised number Exponent $= 0$, mantissa $\neq 0$

We can still do computations using these, but they no longer adhere to the standard's requirements for precision.

Not a number (NaN) Exponent $=$ maximum (e.g. 11111111), mantissa $\neq 0$

These values allow us to recognise when a result is not useful, without aborting an entire computation.

Hidden bit and exponent representation

UNIVERSITY OF CAMBRIDGE

So, we've given up the minimum and the maximum exponents to represent these special values. In single precision, this leaves exponents 1 to 254 for normalised numbers, and for double precision, it leaves 1 to 2046.

In practice, we use a *biased exponent* to allow negative exponents. In single-precision, a stored exponent e represents a true exponent $e - 127$.

This representation is called "excess-127" (single precision) or "excess-1023" (double precision).

Why use it?

Because it means that (for positive numbers, and ignoring NaNs) floating point comparison is the same as integer comparison. Cool!

Why 127 not 128? The committee decided it gave a more symmetric number range.

Digression

IEEE define terms e_{min}, e_{max} delimiting the exponent range and programming languages define constants like

```
#define FLT_MIN_EXP (-125)
#define FLT_MAX_EXP 128
```

whereas on the previous slide I listed the min/max exponent uses as $1.mmmmmm * 2^{-126}$ to $1.mmmmmm * 2^{127}$.

BEWARE: IEEE and ISO C write the above ranges as $0.1mmmmmm * 2^{-125}$ to $0.1mmmmmm * 2^{128}$ (so all p digits are *after* the decimal point) so all is consistent, but remember this if you ever want to use `FLT_MIN_EXP` or `FLT_MAX_EXP`.

I've kept to the more intuitive $1.mmmmm$ form in these notes.

Solved exercises

UNIVERSITY OF CAMBRIDGE

What's the smallest and biggest normalised numbers in single precision IEEE floating point?

Biggest: exponent field is 0..255, with 254 representing 2^{127}. The biggest mantissa is 1.111...111 (24 bits in total, including the implicit leading one) so $1.111...111 \times 2^{127}$. Hence almost 2^{128} which is $2^8 * 2^{120}$ or $256 * 1024^{12}$, i.e. around $3 * 10^{38}$.

`FLT_MAX` from `<float.h>` gives `3.40282347e+38f`.

Smallest? That's easy: $-3.40282347\text{e}{+38}$! OK, I meant smallest positive. I get $1.000...000 \times 2^{-126}$ which is by similar reasoning around 16×2^{-130} or 1.6×10^{-38}.

`FLT_MIN` from `<float.h>` gives `1.17549435e-38f`.

Solved exercises (2)

UNIVERSITY OF CAMBRIDGE

'Denormalised numbers' can range down to
$2^{-23} \cdot 2^{-126} = 2^{-149} \approx 1.401298$e-45, but there is little accuracy at this level.

And the precision of single precision? 2^{23} is about 10^7, so in principle 7sf. (But remember this is for representing a single number, operations will rapidly chew away at this.)

And double precision? `DBL_MAX 1.7976931348623157e+308 and DBL_MIN 2.2250738585072014e-308` with around 16sf.

How many single precision floating point numbers are there?
Answer: 2 signs * 254 exponents * 2^{23} mantissas for normalised numbers plus 2 zeros plus 2 infinities (plus NaNs and denorms ...).

Solved exercises (3)

UNIVERSITY OF CAMBRIDGE

So you mean 0.1 is not exactly representable?

0.1_{10} is $0.00011001100\ldots_2$ a recurring binary number! Check by evaluating the geometric progression:

$$\frac{1}{16} \times ((2^0 + 2^{-1}) + (2^{-4} + 2^{-5}) + \cdots) = \frac{1}{16} \frac{2^0+2^{-1}}{1-(2^{-4})} = \frac{1}{10}.$$

However, we must round it to its nearest single precision IEEE number (note round to nearest) i.e. $\boxed{0}\boxed{011\ 1101\ 1}\boxed{100\ 1100\ 1100\ 1100\ 1100\ 1101}$.

Decoded, this is $2^{-4} \times 1.100\,1100 \cdots 1101_2$, or

$$\frac{1}{16} \times (2^0 + 2^{-1} + 2^{-4} + 2^{-5} + \cdots + 2^{-23}).$$

$$= 0.10000000149$$

This is the source of the Patriot missile bug.

Dividing by a Constant: Example divide by ten.

UNIVERSITY OF CAMBRIDGE

In binary, one tenth is 0.0001100110011....

Divide a 32 bit unsigned integer by 10 by long multiplication by reciprocal using the following hand-crafted code:

```
unsigned div10(unsigned int n)
{ unsigned int q;
  q = (n >> 1) + (n >> 2);  // Ultimately shifts of 4 and 5.
  q = q + (q >> 4);         // Replicate: 0.11 becomes 0.110011
  q = q + (q >> 8);         // 0.110011 becomes 0.1100110011100110011
  q = q + (q >> 16);        // Now have 32 bit's worth of product.
  return q >> 3;            // Doing this shift last of all
                            // gave better intermediate accuracy.
}
```

More on `http://www.hackersdelight.org/divcMore.pdf`

Solved exercises (4)

UNIVERSITY OF CAMBRIDGE

BTW, So how many times does

```
for (f = 0.0; f < 1.0; f += 0.1) { C }
```

iterate? Might it differ if `f` is single/double?

NEVER count using floating point unless you really know what you're doing (and write a comment half-a-page long explaining to the 'maintenance programmer' following you why this code works and why naïve changes are likely to be risky).

Signed zeros, signed infinities

UNIVERSITY OF CAMBRIDGE

Signed zeros can make sense: if I repeatedly divide a positive number by two until I get zero (*'underflow'*) I might want to remember that it started positive, similarly if I repeatedly double a number until I get *overflow* then I want a signed infinity.

However, while differently-signed zeros compare equal, not all 'obvious' mathematical rules still hold:

```
int main() {
  double a = 0, b = -a;
  double ra = 1/a, rb = 1/b;
  if (a == b && ra != rb)
     printf("Ho hum a=%f == b=%f but 1/a=%f != 1/b=%f\n", a,b, ra,rb);
  return 0; }
```

Gives:

```
Ho hum a=0.000000 == b=-0.000000 but 1/a=inf != 1/b=-inf
```

Overflow Exceptions

Overflow is the main potential source of exception.

Using floating point, overflow occurs when an exponent is too large to be stored.

Overflow exceptions most commonly arise in division and multiplication. Divide by zero is a special case of overflow.

But addition and subtraction can lead to overflow. When?

Whether to raise an exception or to continue with a NaN? If we return NaN it will persist under further manipulations and be visible in the output.

Underflow is normally ignored silently: whether the overall result is then poor is down to the quality of the programming.

Exceptions versus infinities and NaNs?

UNIVERSITY OF CAMBRIDGE

The alternatives are to give either a wrong value, or an exception.

An infinity (or a NaN) propagates 'rationally' through a calculation and enables (e.g.) a matrix to show that it had a problem in calculating some elements, but that other elements can still be OK.

Raising an *exception* is likely to abort the whole matrix computation and giving wrong values is just plain dangerous.

The most common way to get a NaN is by calculating 0.0/0.0 (there's no obvious 'better' interpretation of this) and library calls like `sqrt(-1)` generally also return NaNs (but results in scripting languages can return $0 + 1i$ if, unlike Java, they are untyped or dynamically typed and so don't need the result to fit in a single floating point variable).

IEEE 754 History

UNIVERSITY OF CAMBRIDGE

Before IEEE 754 almost every computer had its own floating point format with its own form of rounding – so floating point results differed from machine to machine!

The IEEE standard largely solved this (in spite of mumblings "this is too complex for hardware and is too slow" – now obviously proved false). In spite of complaints (e.g. the two signed zeros which compare equal but which can compare unequal after a sequence of operators) it has stood the test of time.

However, many programming language standards allow intermediate results in expressions to be calculated at higher precision than the programmer requested so

`f(a*b+c)` and

`{ float t=a*b; f(t+c); }` may call `f` with different values. (Sigh!)

IEEE arithmetic

IEEE basic operations ($+, -, *, /$ are defined as follows):

> Treat the operands (IEEE values) as precise, and do perfect mathematical operations on them. Round this mathematical value to the *nearest* representable IEEE number and store this as the result. In the event of a tie, choose the value with an even (i.e. zero) least significant bit.

This is the binary version of Banker's Rounding: $1.5 \to 2.0$, $4.5 \to 4.0$

[This last rule is statistically fairer than the "round down 0–4, round up 5–9" which you learned in school. *Don't be tempted to believe the exactly 0.50000 case is rare!*]

IEEE Rounding

IEEE requires there to be a global flag which can be set to one of 4 values:

Unbiased which rounds to the nearest value. If the number falls midway it is rounded to the nearest value with an even (zero) least significant bit. This mode is required to be default.

Towards zero

Towards positive infinity

Towards negative infinity

Be very sure you know what you are doing if you change the mode, or if you are editing someone else's code which exploits a non-default mode setting.

Other mathematical operators?

Other mathematical operators are typically implemented in libraries. Examples are `sin`, `sqrt`, `log` etc. It's important to ask whether implementations of these satisfy the IEEE requirements: e.g. does the `sin` function give the nearest floating point number to the corresponding perfect mathematical operation's result when acting on the floating point operand treated as perfect? [This would be a perfect-quality library with error within 0.5 ulp (Unit in the Last Place) and still a research problem for most functions.]

Or is some lesser quality offered? In this case a library (or package) is only as good as the vendor's careful explanation of what error bound the result is accurate to. ± 1 ulp is excellent.

But remember (see 'ill-conditionedness') that a more important practical issue might be how a change of 1 ulp on the input(s) affects the output – and hence how input error bars become output error bars.

The java.lang.Math libraries

UNIVERSITY OF CAMBRIDGE

Java has quite well-specified math routines, e.g. for `asin()` "arc sine":

Returns the arc sine of an angle, in the range of -pi/2 through pi/2. Special cases:

- If the argument is NaN or its absolute value is greater than 1, then the result is NaN.
- If the argument is zero, then the result is a zero with the same sign as the argument.

A result must be within 1 ulp of the correctly rounded result
(perfect accuracy requires result within 0.5 ulp).

Results must be **semi-monotonic**:
(i.e. given that arc sine is monotonically increasing, this condition requires that $x < y$ implies $\texttt{asin}(x) \leq \texttt{asin}(y)$)

Unreported Errors in Floating Point Computations

Overflow is reported (exception or NaN).

When we use floating point, unreported errors (w.r.t. perfect mathematical computation) essentially arise from two sources:

- **quantisation errors** arising from the inexact representation of constants in the program and numbers read in as data. (Remember even 0.1 in decimal cannot be represented exactly in as an IEEE value, just like 1/3 cannot be represented exactly as a finite decimal.)
- **rounding errors** produced by (in principle) every IEEE operation.

These errors build up during a computation, and we wish to be able to get a bound on them (so that we know how accurate our computation is).

(Use all of the error analysis techniques you learned earlier in the course.)

Machine Epsilon

UNIVERSITY OF CAMBRIDGE

Machine epsilon (ϵ_m) is defined as the difference between 1.0 and the smallest *representable* number which is greater than one, i.e. 2^{-23} in single precision, and 2^{-52} in double (in both cases $\beta^{-(p-1)}$). ISO 9899 C says:

> The difference between 1 and the least value greater than 1 that is representable in the given floating point type.

i.e. machine epsilon is 1 ulp for the representation of 1.0.

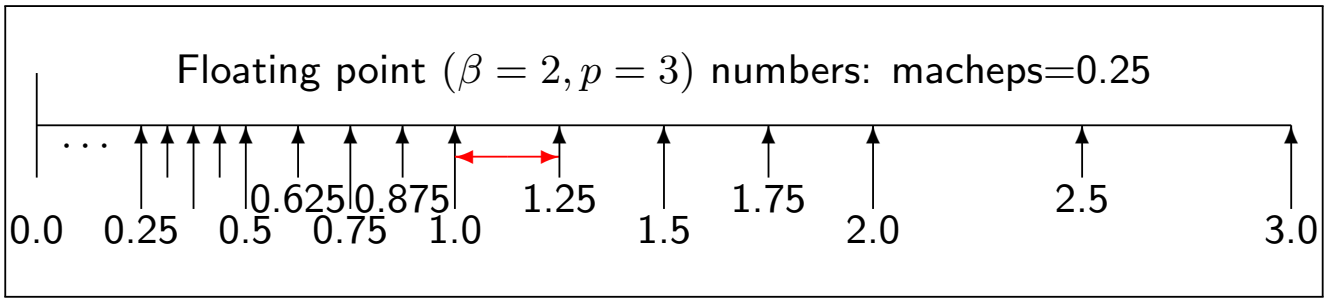
For IEEE arithmetic, the C library `<float.h>` defines

```
#define FLT_EPSILON         1.19209290e-7F
#define DBL_EPSILON         2.2204460492503131e-16
```

Machine Epsilon (2)

UNIVERSITY OF CAMBRIDGE

Machine epsilon is useful as it gives an upper bound on the relative error caused by getting a floating point number wrong by 1 ulp, and is therefore useful for expressing errors independent of floating point size.



(The relative error caused by being wrong by 1 ulp *can* be up to 50% smaller than this, consider 1.5 or 1.9999.)

Machine Epsilon (3)

UNIVERSITY OF CAMBRIDGE

Some sources give an alternative (bad) definition:

Microsoft MSDN documentation (Feb 2009):

Constant	Value	Meaning
FLT_EPSILON	1.192092896e-07F	Smallest such that 1.0+FLT_EPSILON !=1.0

The value is right by the C standard, but the explanation inconsistent.

With rounding-to-nearest, the intermediate value only needs to be closer to $1 + \epsilon_m$ than 1. Whoops:

```
  float one = 1.0f, xeps = 0.7e-7f;  // Smaller than FLT_EPSILON!
  printf("%.7e + %.7e = %.7e\n", one, xeps, (float)(xeps+one));
===>>>>  1.0000000e+00 + 6.9999999e-08 = 1.0000001e+00
```

Machine Epsilon (4) – Negative Epsilon

UNIVERSITY OF CAMBRIDGE

We defined machine epsilon as the difference between 1.0 and the smallest *representable* number which is greater than one.

What about the difference between 1.0 and the greatest *representable* number which is smaller than one?

In IEEE arithmetic this is exactly 50% of machine epsilon.

Why? Let's illustrate with precision of 5 binary places (4 stored). One is $2^0 \times 1.0000$, the next smallest number is $2^0 \times 0.111111111\ldots$ truncated to fit. But when we write this normalised it is $2^{-1} \times 1.1111$ and so its ulp represents only half as much as the ulp in $2^0 \times 1.0001$.

Range reduction

UNIVERSITY OF CAMBRIDGE

Consider implementing the function $\sin x$ using its Taylor expansion:

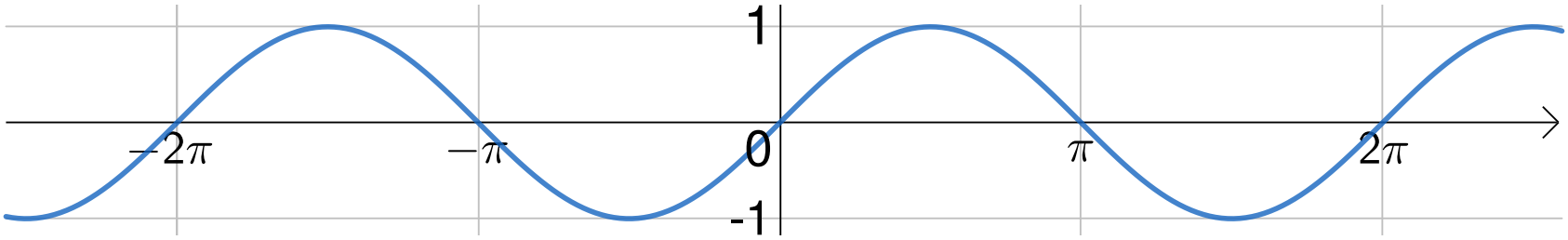
$$\sin x \quad = \quad x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots$$

If we are given an x with large magnitude:

- It will take longer for the terms to start shrinking, so we will need to compute more of them to get a suitably precise result.
- The intermediate terms will be much larger than the final result, so there will necessarily be loss of significance as the terms cancel each other.

Range reduction

Fortunately, $\sin x$ is periodic, so it is fairly straightforward to pre-process x to give something which behaves nicely.



- The obvious approach is to use modular arithmetic to bring x into the range $[-\pi, \pi)$.
- With a little thought, you may notice that $[0, \frac{\pi}{2}]$ is sufficient. This exploits reflectional symmetry, so requires more-complex pre-processing, and also post-processing of the result.
- $[0, \frac{\pi}{4}]$ is even better – how would that work?

Range reduction

UNIVERSITY OF CAMBRIDGE

What if I make sure I don't use large x in the first place?

Easier said than done. It's easy to miss e.g. $\sin(2\pi ft)$ in a loop, with t incrementing on every timestep.

There are also some situations where the function is not periodic, so large x make more sense.

Some series even **require** range reduction in order to guarantee convergence. e.g.:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots \quad \text{for } |x| < 1$$

`sin 1e40`

UNIVERSITY OF CAMBRIDGE

`xcalc` says:

$$\sin 1e40 = 0.3415751$$

Does anyone believe this result? (Try your calculator/programs on it!)

`sin 1e40`

UNIVERSITY OF CAMBRIDGE

The answer given by `xcalc` is totally bogus. Why?

10^{40} is stored (like all numbers) with a relative error of around machine epsilon. (So changing the stored value by 1 ulp results in a relative error of around $machine\_epsilon$, which is an absolute error of $10^{40} \times machine\_epsilon$.) Even for `double` (16sf), this absolute error of representation is around 10^{24}. But the `sin` function cycles every 2π. So we can't even represent which of many billions of cycles of sine that 10^{40} should be in, let alone whether it has any sig.figs.!

On a decimal calculator 10^{40} is stored accurately, but I would need π to 50sf to have 10sf left when I have range-reduced 10^{40} into the range $[0, \pi/2]$.