369 Computational Science

Compiled, collated and written by David Welch
Sections 1-9 are based on slides by Georgy Gimel'farb, sections 16–18 are based on the book Biological Sequence Analysis by Durbin et al. 2010.
Other sections are incompletely referenced so presence in these notes is no claim of authorship.

Contents

1 Introduction

This course is aimed at introducing computer scientists to uses of computers and computational techniques in other areas of science. The number of ways that computers are used in the sciences are many, varied and often extremely sophisticated. The focus of this course will be on "computational science" which involves constructing mathematical models that can be simulated, analysed and solved using computational methods.

The course is split into two parts: in the first 3-4 weeks, we'll look at techniques for finding the roots of equations, solving systems of linear equations and decomposing matrices. These techniques are basic to areas of research known as computational engineering, numerical analysis and applied linear algebra.

In the remaining 8-9 weeks, we'll turn to computational biology, with a focus on bioinformatics and phylogenetics. There, we see how a wide range of computational and mathematical techniques have revolutionised an area of science and allowed us to analyse and interpret huge amounts of genetic data. This area of study has helped us better understand, among other things, the basic workings of life, our evolutionary history, the causes of inherited diseases and the spread of infectious disease.

From a computational point of view, computational biology is a fascinating and active area of research. The techniques we'll study in this part of the course include stochastic and probabilistic modelling, simulation, dynamic programming, estimation and inference.

CS 369 is more mathematical than many CS courses. This is unavoidable given the subject matter. We assume that students have some background in discrete mathematics (matrices, graphs, linear equations) and continuous mathematics (functions, derivatives, integration), and an understanding of basic probability (discrete and continuous random variables, expectation, conditional probability) . However, we recognise that students come to this course from a variety of backgrounds so will provide explanations from quite a basic level in most cases. We do assume that students have a solid foundation in programming in one of Java, C++, Python, Matlab or R.

2 Mathematical modelling and why we need computers

Mathematical models attempt to precisely describe a system in order to better understand it. A model is usually based on observing the system and is often structured to answer a particular question. It is not an exact replica of the system and is not merely a description of the observations. Recorded observations of the system are known as data. Coupled with data, the model allows us to infer unobserved properties of the model (such as model parameters) and predict future outcomes. Careful comparison of outcomes predicted by the model with data (actual outcomes) tell us how accurate the model is and where it needs to be refined.

This process of modelling and observation has, arguably, been used for thousands of years and certainly for hundreds. The complexity of the models we create and study is somewhat determined by our ability to interpret and “solve” them. Before computers, we were largely limited to using models that were analytically tractable — that is, models for which closed form solutions could be found — or for which good approximations could be made by hand. Our ability to fit models to data was severely limited by our human limitations of collecting, storing and processing information by hand.

With the advent of computers, both of these limitations have eased considerably. It is now possible to collect and store massive amounts of data. For example, Genbank, which stores genetic nucleotide sequences contains over 204 billion nucleotide bases in more than 189 million sequences as at the end of 2015, while CERN’s Large Hadron Collider produces 30 PB ($= 30 \times 10^6$ GB) of data annually. And fast computers allow us process this data and to make almost arbitrarily good approximations to models that are far more complex than could be tackled by hand.

However, even with all the data and computing power in the world we need to be careful to propose useful models and tackle them with efficient techniques if we are to make progress in answering questions that interest us. Bad models, bad data or bad computational techniques could all derail our quest for understanding. In this course, we aim to teach good computational techniques and give some insight into some basic modelling and data analysis techniques that will help to tackle and answer a range of interesting questions.

2.1 Why we need to be clever about our computing

Mathematical problems can be classed into problems that are *well-posed* and *ill-posed.* A problem is *well-posed* if

1. A solution exists
2. the solution is unique
3. A small change in the initial condition induces only a small change in the solution

A problem that is not well-posed is ill-posed.

We are interested in the last criterion which can be termed *sensitivity.* Suppose our problem has inputs x and has a solution (or output) y. An *insensitive* or well-conditioned problem is when a change in x causes a change in y that is of similar relative size. A *sensitive* or *ill-conditioned* problem is one in which the change in solution/output can be large relative the the change in input.

Based on this idea, define the *condition number* by

$$cond = \frac{|\text{relative change in solution}|}{|\text{relative change in input data}|} = \left|\frac{\Delta y/y}{\Delta x/x}\right|.$$

Thus a problem is *ill-conditioned* is $cond \gg 1$.

Example: what is the condition number when we evaluate a function $y = f(x)$ at an approximation of x, $\hat{x} = x + \Delta x$, rather than at the true value x?

Solution:

$$cond = \left|\frac{\Delta y/y}{\Delta x/x}\right| = \left|\frac{f(x+\Delta x) - f(x)/f(x)}{\Delta x/x}\right| = \left|\frac{f(x+\Delta x) - f(x)}{\Delta x}\frac{x}{f(x)}\right| \approx \left|\frac{xf'(x)}{f(x)}\right|.$$

So, depending on the function f and the input x, we could get very large condition numbers. $\square$

Example: What is the condition number for the functions $f(x) = x^n$ and $f(x) = e^x$?

Solution: From above, the condition number is

$$cond \approx \left|\frac{xf'(x)}{f(x)}\right|.$$

When $f(x) = x^n$, $f'(x) = nx^{n-1}$, so

$$cond = \left|\frac{xf'(x)}{f(x)}\right| = \left|\frac{x.nx^{n-1}}{x^n}\right| = \left|\frac{nx^n}{x^n}\right| = |n|.$$

So as the degree of the polynomial increases, the problem becomes increasingly ill-conditioned.

Similarly, when $f(x) = e^x$, $f'(x) = e^x$ so

$$cond = \left|\frac{xf'(x)}{f(x)}\right| = \left|\frac{x.e^x}{e^x}\right| = |x|.$$

In this case, the condition number depends on the input argument, x. If x is very large, the problem can be considered ill-conditioned. $\square$

What does this mean for computation? In nearly all our computations, we will replace exact formulations with approximations. For example, we approximate continuous quantities with discrete quantities, we are limited in size by floating point representation of numbers which often necessitates rounding or truncation and introduces (hopefully) small errors. If we are not careful, these error will be amplified within an ill-conditioned problem an produce inaccurate results.

This leads us to consider the *stability* and *accuracy* of our algorithms.

An algorithm is *stable* if the result is relatively unaffected by perturbations during computation. This is similar to the idea of conditioning of problems.

An algorithm is *accurate* is the competed solution is close to the true solution of the problem. Note that a stable algorithm could be inaccurate.

We seek to design and apply stable and accurate algorithms to find accurate solutions to well-posed problems (or to find ways of transforming or approximating ill-posed problems by well-posed ones).

3 Approximating a function by a Taylor series

First, a little notation. A real-valued function $f : \mathbb{R} \to \mathbb{R}$ is a function f that takes a real number argument, $x \in \mathbb{R}$,and returns a real number, $f(x) \in \mathbb{R}$. We write $f'(x)$ to denote the derivative of f, so $f'(x) = \frac{df}{dx}$, and $f''(x)$ is the second derivative (so $f'(x) = \frac{d}{dx}(\frac{df}{dx}) = \frac{d^2 f}{dx^2}$) and $f^{(k)}(x)$ is the kth derivative of f evaluated at x.

As we have just seen, simply evaluating a real valued function can be prone to instability (if the function has a large condition number). For that reason, approximations of functions are often used. The most common method of approximating the real-valued function $f : \mathbb{R} \to \mathbb{R}$ by a simpler function is to use the Taylor series representation for f.

The Taylor series has the form of a polynomial where the coefficients of the polynomial are the derivatives of f evaluated at a point. So long as all derivatives of the function exists at the point $x = a$, $f(x)$ can be expressed in terms of of the value of the function and it's derivatives at a as:

$$f(x) = f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2!}f''(a) + \ldots + \frac{(x-a)^k}{k!}f^{(k)}(a) + \ldots$$

This can be written more compactly as

$$f(x) = \sum_{k=0}^{\infty} \frac{(x-a)^k}{k!} f^{(k)}(a),$$

where $f^{(0)} = f$ and $0! = 1$ by definition.

This is known as the *Taylor series* for f about a. It is valid for x "close" to a (strictly, within the "radius of convergence" of the series). When $a = 0$, the Taylor series is known as a *Maclaurin series.*

This is an infinite series (the sum contains infinitely many terms) so cannot be directly computed. In practice, we truncate the series after n terms to get the *Taylor polynomial of degree* n centred at a, which we denote $\hat{f}_n(x; a)$:

$$f(x) \approx \hat{f}_n(x; a) = \sum_{k=0}^{n} \frac{(x-a)^k}{k!} f^{(k)}(a).$$

This is an approximation of f that can be readily calculated so long as the first n derivatives of f evaluated at a can be calculated. The approximation can be made arbitrarily accurate by increasing n. The quality of the approximation also depends on the distance of x from a — the closer x is to a, the better the approximation.

Example: Find the Taylor approximation of $f(x) = \exp(x) = e^x$ for values of x close to 0.

Solution: The kth derivative of $f(x) = e^x$ is simply e^x for all k. Since we want values of x close to 0, find the Taylor series about $a = 0$ (the Maclaurin series). Then

$$\widehat{f}_n(x;0) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{k=0}^{n} \frac{x^k}{k!}$$

This series converges to e^x everywhere: $\lim_{\to\infty} \widehat{e}_n(x) = e^x$. The quality of the approximation for various values of n and x are studied in the table below.

n	1	2	3	4	...	true value e^x
$\widehat{f}_n(x=1;0)$	2.0000	2.5000	2.6667	2.7083	...	2.7183
Relative error	0.26	0.08	0.019	0.0037		
$\widehat{f}_n(x=2;0)$	3.0000	5.0000	6.3333	7.0000	...	7.3891
Relative error	0.59	0.32	0.14	0.053		

Notice that the error is smaller for x close to a and decreases as n, the number of terms in the polynomial increases. □

4 Finding the roots of equations

Given a real valued function $f : \mathbb{R} \to \mathbb{R}$, a fundamental problem is to find the values of x for which $f(x) = 0$. This is known as finding the roots of f. The problem crops up again and again and many problems can be reformulated as this problem. For example, if the trajectory of one object is described by $h(x)$, while another object has trajectory $g(x)$, then the two objects intercept one another exactly when $f(x) = g(x) - h(x) = 0$.

Another common application is when we wish to find the minimum or maximum value that a function takes. We know from basic calculus that if f is the derivative of some function F, then $F(x)$ takes its maximum or minimum values when $f(x) = 0$. Thus finding the maximum value of F is a matter of finding the roots of f.

We will consider two simple yet effective root finding algorithms. The bisection method and Newton's method. In both cases, we assume that $f : \mathbb{R} \to \mathbb{R}$ is continuous.

4.1 Bisection method

We want to find x^* such that $f(x^*) = 0$. Let $\text{sign}(f(x)) = -1$ if $f(x) < 0$ and $\text{sign}(f(x)) = 1$ if $f(x) > 0$. The bisection method proceeds as follows:

- **Initialise:** Choose initial guesses a, b such that $\text{sign}f(a) \neq \text{sign}f(b)$
- **Iterate** until the absolute difference $|a - b| \approx 0$
  - Calculate $c = \frac{a+b}{2}$
  - If $\text{sign}f(a) = \text{sign}f(c)$, then $a \leftarrow c$; otherwise $b \leftarrow c$

The method provides slow but sure convergence.

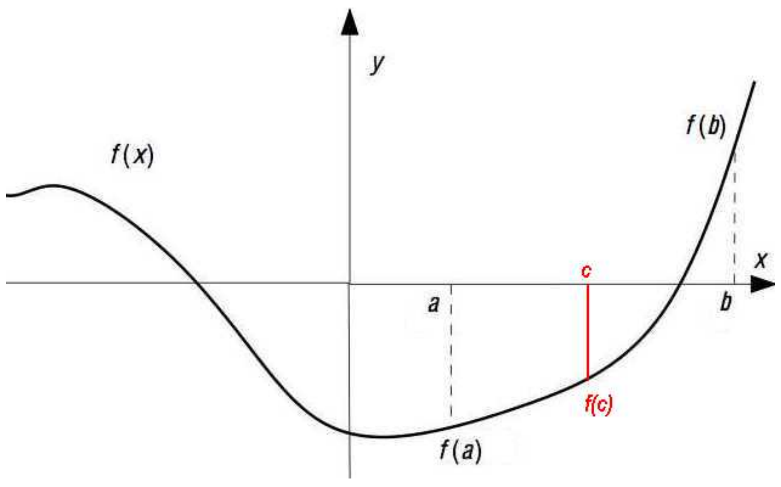


Figure 1: The idea behind the bisection method. Here, $\text{sign} f(a) = \text{sign} f(c)$ so at the next iteration we will set $a \leftarrow c$. The algorithm stops when a and b are sufficiently close to each other.

4.2 Newton's method

This method is also known as the Newton-Raphson method and is based on the approximation first two terms of the Taylor series expansion. Recall that we want to find x such that $f(x) = 0$. From the first two terms of the Taylor series of $f(x)$, we know that $f(x) \approx f(a) + (x-a)f'(a)$. If $f(x)$ is zero, then the expression on the right hand side to 0 and solve for x to get

$$f(a) + (x-a)f'(a) = 0 \implies x = a - \frac{f(a)}{f'(a)}.$$

We can treat this iteratively, starting at x_0, and finding $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$. This leads to the algorithm:

- **Initialise:** Choose x_0 as an initial guess.
- **Iterate** until the absolute difference $|x_i - x_{i-1}| \approx 0$
  - Set $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$.

Example: $f(x) = x^2 - 2$.
Solution: $f'(x) = 2x$ so

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{x_i^2 - 2}{2x_i} = \frac{x_i}{2} + \frac{1}{x_i}.$$

See slide for example starting at $x_0 = 0.5$. $\square$

Compare with bisection method: Start at $a = 1/2$ and $b = 2$ to get to $a = 1.34375$ and $b = 1.4375$ at the fifth step. This produces an absolute error of 0.02688 or 1.9%. The absolute error in the Newton case is 0.00020 which is 2 orders of magnitude smaller.

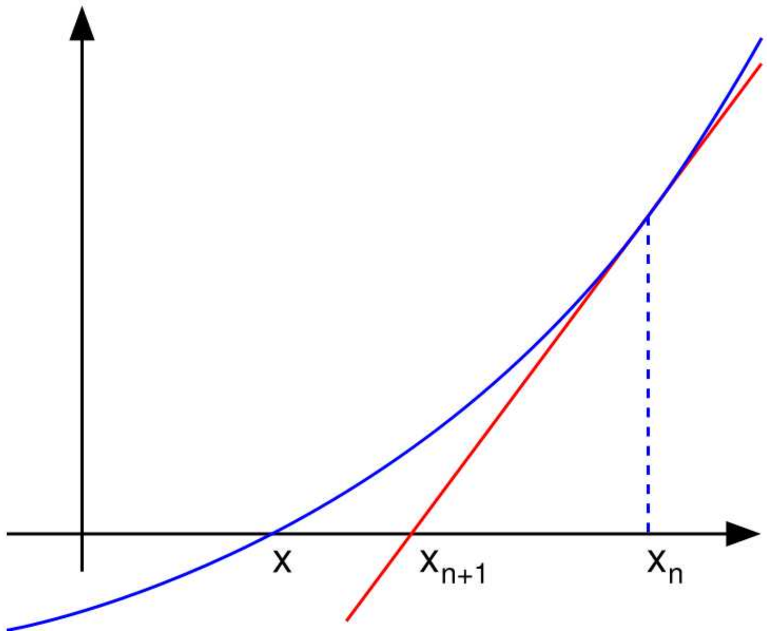


Figure 2: The idea behind the Newton's method. Starting at x_n, we find the tangent line (red) and calculate the point it intercepts the x-axis. This point of intercept is x_{n+1}.

There are, of course, many other root finding methods. We list a few of them here (not examinable).

Secant method (`http://en.wikipedia.org/wiki/Secant_method`):

- Newton's method with a finite difference instead of the derivative
- Neither computation, nor existence of a derivative is required
- However, the convergence is slower (approximately, $\alpha = 1.6$)

False position method (`http://en.wikipedia.org/wiki/False_position_method`):

- Always retains one point on either side of the root
- Faster than the bisection and more robust than the secant method

Muller's method (`http://en.wikipedia.org/wiki/Muller's_method`):

- Quadratic (instead of linear) interpolations
- Faster convergence than with the secant method
- Roots may be complex (in addition to reals)

5 Numerical linear algebra

Numerical linear algebra is one of the cornerstones of modern mathematical modelling. Topics as important as solving systems of ordinary differential equations (arising in engineering, economics, physics, biotech, etc), to network analysis (telecoms, sociologic, epidemiology), internet search, data mining and many more rely on linear algebra.

These days, applied linear algebra and numerical linear algebra are virtually interchangeable — problems of all sizes are routinely solved numerically and rely on a wealth of mathematical and computational insight.

We'll start out with a brief review of topics that you should be somewhat familiar with.

5.1 Review

Let $\mathbf{a} = \begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix}$ $\mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$ be vectors. The *inner* or *dot product* of **a** and **b** is the scalar $c = \mathbf{a} \bullet \mathbf{b} \equiv \mathbf{a}^\mathsf{T}\mathbf{b} = \sum_{i=1}^{n} a_i b_i$. The dot product is also called *multiplication* of vectors.

The *norm* or *magnitude* of a vector **a** is $||\mathbf{a}|| = \sqrt{a \cdot a} = \sqrt{\mathbf{a}^\mathsf{T}\mathbf{a}} = \sqrt{a_1^2 + \ldots a_n^2}$.

The *product* of an $m \times n$ matrix $\mathbf{A} = \begin{bmatrix} A_{11} & \ldots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{m1} & \ldots & A_{mn} \end{bmatrix}$ and an $n \times 1$ (n-dimensional) vector $\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$ is the m-dimensional vector $\mathbf{y} = \mathbf{A}\mathbf{x}$ with the elements $y_i = \sum_{j=1}^{m} A_{ij} x_j$.

The *product* of a $k \times m$ matrix **A** and an $m \times n$ matrix **B** is the $k \times n$ matrix $\mathbf{C} = \mathbf{AB}$ with the elements $C_{ij} = \sum_{\alpha=1}^{m} A_{i,\alpha} B_{\alpha,j}$

The *outer product* of an m-dimensional vector **a** with an n-dimensional vector **b** is the $m \times n$ matrix

$$\mathbf{a}\mathbf{b}^\mathsf{T} \equiv \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} \begin{bmatrix} b_1 & b_2 & \ldots & b_n \end{bmatrix} = \begin{bmatrix} a_1b_1 & a_1b_2 & \ldots & a_1b_n \\ a_2b_1 & a_2b_2 & \ldots & a_2b_n \\ \vdots & \vdots & \ddots & \vdots \\ a_mb_1 & a_mb_2 & \ldots & a_mb_n \end{bmatrix}$$

The *identity matrix* of size n, $\mathbf{I}_n$, is the $n \times n$ matrix with (i,j)th entry $= 0$ if $i \neq j$ and 1 if $i = j$.

The *inverse* of a square matrix **A** of size n is the square matrix $\mathbf{A}^{-1}$ such that $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}_n = \mathbf{A}^{-1}\mathbf{A}$. When such a matrix exists, A is called *invertible* or *non-singular.* **A** is *singular* if no inverse exists. Finding the inverse of **A** is typically difficult.

The *determinant* of an $n \times n$ matrix $\mathbf{A}$, written $det(\mathbf{A}) = \begin{vmatrix} A_{11} & \dots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{n1} & \dots & A_{nn} \end{vmatrix}$, is given by a somewhat complex formula that we need not reproduce here (look it up at `http://en.wikipedia.org/wiki/Determinant`). For $n = 2$, $det(A) = A_{11}A_{22} - A_{21}A_{12}$. For $n = 3$, $det(A) = A_{11}A_{22}A_{33} - A_{31}A_{22}A_{13} + A_{12}A_{23}A_{31} - A_{32}A_{23}A_{11} + A_{13}A_{21}A_{32} - A_{33}A_{21}A_{12}$.

Example: Find the determinant of $\mathbf{A} = \begin{pmatrix} 3 & 5 \\ 1 & -1 \end{pmatrix}$.

Solution: From above, $det(\mathbf{A}) = |\mathbf{A}| = 3.-1 - 1.5 = -3 - 5 = -8$. $\square$

It is worth recalling a few properties of the determinant (as listed on the wiki page):

- $det(\mathbf{I}) = 1$
- $det(A^T) = det(A)$ (transposing the matrix does not affect the determinant)
- $det(A^{-1}) = \frac{1}{det(A)}$ (the determinant of the inverse is the inverse of the determinant)
- For A, B square matrices of equal size, $det(AB) = det(A)det(B)$
- $det(cA) = c^n\, det(A)$ for any scalar c
- If A is triangular (so has all zeros in the upper or lower triangle) then $det(A) = \prod_{i=1}^{n} A_{ii}$.

An *eigenvector* of the square matrix $\mathbf{A}$ is a non-zero vector $\mathbf{e}$ such that $\mathbf{Ae} = \lambda\mathbf{e}$ for some scalar λ. λ is known as the *eigenvalue* of $\mathbf{A}$ corresponding to $\mathbf{e}$. Note that λ may be 0. So the effect of multiplying e by A is simply to scale e by the corresponding scalar λ.

The determinant can be used to find the eigenvalues of $\mathbf{A}$: they are the roots of the *characteristic polynomial* $p(\lambda) = \det(\mathbf{A} - \lambda\mathbf{I}_n)$ where $\mathbf{I}_n$ is the identity matrix.

Example: Find the eigenvalues of $\mathbf{A} = \begin{pmatrix} 3 & 5 \\ 1 & -1 \end{pmatrix}$.

Solution: We need to solve $p(\lambda) = \det(\mathbf{A} - \lambda\mathbf{I}_2) = 0$.

$$\begin{aligned} |\mathbf{A} - \lambda\mathbf{I}_2| &= \left| \begin{pmatrix} 3 & 5 \\ 1 & -1 \end{pmatrix} - \lambda \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \right| \\ &= \begin{vmatrix} 3-\lambda & 5 \\ 1 & -1-\lambda \end{vmatrix} \\ &= (3-\lambda)(-1-\lambda) - 5 \\ &= -\lambda^2 - 2\lambda - 8 \\ &= (\lambda+2)(\lambda-4) \end{aligned}$$

which is zero when $\lambda = 4$ or $\lambda = -2$. So the eigenvalues of $\mathbf{A}$ are $\lambda = 4$ and $\lambda = -2$. $\square$

Example: Find the eigenvector of $\mathbf{A} = \begin{pmatrix} 3 & 5 \\ 1 & -1 \end{pmatrix}$ corresponding to the eigenvalue $\lambda = -2$.

Solution: The eigenvector $\mathbf{e}$ corresponding to $\lambda = -2$ satisfies the equation $\mathbf{Ae} = -2\mathbf{e}$. That is,

$$\begin{pmatrix} 3 & 5 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} e_1 \\ e_2 \end{pmatrix} = -2 \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}.$$

This is the system of linear equations

$$\begin{aligned} 3e_1 + 5e_2 &= -2e_1, && (1) \\ e_1 - e_2 &= -2e_2. && (2) \end{aligned}$$

Rearranging either equation, we get $e_1 = -e_2$, so both equations are the same. We thus fix $e_1 = 1$ and the eigenvector associated with $\lambda = -2$ is $\mathbf{e} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$. Notice that the choice to fix $e_1 = 1$ was arbitrary. We could choose any value so, strictly, $\mathbf{e} = c \begin{pmatrix} 1 \\ -1 \end{pmatrix}$ for any $c \neq 0$. Often, c is chosen so that $\mathbf{e}$ is normalised (see below). In this case, choose $c = 1/\sqrt{2}$ to normalise $\mathbf{e}$. □

Vectors a and b are *orthogonal* if the dot product $a^T b = 0$. Orthogonal generalises the of the idea of the perpendicular. In particular, a set of vectors $\{\mathbf{e}_1, \ldots, \mathbf{e}_n\}$ is *mutually orthogonal* if each pair of vectors e_i, e_j is orthogonal for $i \neq j$.

A vector $\mathbf{e}_i$ is normalised if $\mathbf{e}_i^T e_i = 1$.

A set of vectors that is mutually orthogonal and has each vector normalise is called *orthonormal.*

Any symmetric, square matrix $\mathbf{A}$ of size n has exactly n eigenvectors that are mutually orthogonal.

Any square matrix A of size n that has n mutually orthogonal eigenvectors can be represented via the *eigenvector representation* as follows:

$$\mathbf{A} = \sum_{i=1}^{n} \lambda_i \underbrace{\mathbf{e}_i \mathbf{e}_i^{\mathsf{T}}}_{\mathbf{U}_i}$$

where $\mathbf{U}_i = \mathbf{e}_i \mathbf{e}_i^{\mathsf{T}}$ is an $n \times n$ matrix.

The *Range*, range($\mathbf{A}$), or span of an $m \times n$ matrix $\mathbf{A}$ is the set of vectors $\mathbf{y} \in \mathbb{R}^m$ such that $\mathbf{y} = \mathbf{Ax}$ for some $\mathbf{x} \in \mathbb{R}^n$. The range is also referred to as the *column space* of $\mathbf{A}$ as it is the space of all linear combinations of the columns of $\mathbf{A}$.

The *Nullspace*, null($\mathbf{A}$), of an $m \times n$ matrix $\mathbf{A}$ is the set of vectors $\mathbf{x} \in \mathbb{R}^n$, such that $\mathbf{Ax} = \mathbf{0} \in \mathbb{R}^m$

The *Rank*, rank($\mathbf{A}$), of an $m \times n$ matrix $\mathbf{A}$ is the dimension of the range of $\mathbf{A}$ or of the column space of $\mathbf{A}$. rank($\mathbf{A}$) $\leq \min\{m, n\}$.

5.2 Review of eigenvectors and eigenvalues

- λ is an eigenvalue of $\mathbf{A}$ if determinant $|\mathbf{A} - \lambda\mathbf{I}| = 0$
- This determinant is a polynomial in λ of degree n: so it has n roots $\lambda_1, \lambda_2, \ldots, \lambda_n$
- Every symmetric matrix $\mathbf{A}$ has a full set (basis) of n orthogonal unit eigenvectors $\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n$
- No algebraic formula for the polynomial roots for $n > 4$
  - Thus, the eigenvalue problem needs own special algorithms
  - Solving the eigenvalue problem is harder than solving $\mathbf{Ax} = \mathbf{b}$
- Determinant $|\mathbf{A}| = \prod_{i=1}^{n} \lambda_i = \lambda_1\lambda_2\cdots\lambda_n$ (the product of eigenvalues)
- The *trace* of a matrix is the sum of the diagonal elements. That is, $\text{trace}(\mathbf{A}) = \sum_{i=1}^{n} a_{ii} = a_{11} + a_{22} + \ldots + a_{nn}$.
- It turns out that $\text{trace}(\mathbf{A}) = \sum_{i=1}^{n} \lambda_i = \lambda_1 + \lambda_2 + \ldots + \lambda_n$ (the sum of eigenvalues)
- $\mathbf{A}^k = \underbrace{\mathbf{A}\cdots\mathbf{A}}_{k \text{ times}}$ has the same eigenvectors as $\mathbf{A}$: e.g. for $\mathbf{A}^2$

$$\mathbf{Ae} = \lambda\mathbf{e} \;\Rightarrow\; \mathbf{AAe} = \lambda\mathbf{Ae} = \lambda^2\mathbf{e}$$

- Eigenvalues of $\mathbf{A}^k$ are $\lambda_1^k, \ldots, \lambda_n^k$
- Eigenvalues of $\mathbf{A}^{-1}$ are $\frac{1}{\lambda_1}, \ldots, \frac{1}{\lambda_n}$

Example: Find the eigenvalues and eigenvectors of $\mathbf{A} = \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}$

Solution: First, find the eigenvalues of $\mathbf{A}$ by solving

$$|\mathbf{A} - \lambda\mathbf{I}| = \begin{vmatrix} 2-\lambda & -1 \\ -1 & 2-\lambda \end{vmatrix} = \lambda^2 - 4\lambda + 3 = (\lambda - 1)(\lambda - 3) = 0.$$

So the eigenvalues are $\lambda_1 = 1$ and $\lambda_2 = 3$

The eigenvector associated with $\lambda_1 = 1$ is $\mathbf{e}_1$ and satisfies $\mathbf{Ae}_1 = \lambda_1\mathbf{e}_1$. Putting $\mathbf{e}_1 = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$ we need to solve

$$\begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}.$$

The second row gives $-x_1 + 2y_1 = y_1$, so $y_1 = x_1$. So fix $x_1 = 1$ and $e_1 = c \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ for any $c \neq 0$. If we choose c so that e_1 is normalised, $\mathbf{e}_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$. A similar argument shows $\mathbf{e}_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix}$.

Before leaving this example, it is worth looking at some of the properties of the eigenvalues of $\mathbf{A}$:

- Determinant $\det \mathbf{A} \equiv |\mathbf{A}| = 4 - 1 = 3 \iff \lambda_1 \cdot \lambda_2 \equiv 1 \cdot 3 = 3$
- $\text{trace}(\mathbf{A}) = 2 + 2 = 4 \iff \lambda_1 + \lambda_2 \equiv 1 + 3 = 4$
- Inverse matrix $\mathbf{A}^{-1} = \frac{1}{3} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$: eigenvalues $\lambda_1 = \frac{1}{3}$ and $\lambda_2 = 1$
- Matrix $\mathbf{A}^2 = \begin{bmatrix} 5 & -4 \\ -4 & 5 \end{bmatrix}$: eigenvalues $\lambda_1 = 1$ and $\lambda_2 = 9$
- Matrix $\mathbf{A}^3 = \begin{bmatrix} 14 & -13 \\ -13 & 14 \end{bmatrix}$: eigenvalues $\lambda_1 = 1$ and $\lambda_2 = 27$

$\square$

5.3 Review of systems of linear equations

A linear equation in n unknowns $x_1, \ldots, x_n$ is of the form $a_1x_1 + \ldots a_nx_n = b$. Given m such equations, we can write the ith equation as $a_{i1}x_1 + \ldots a_{in}x_n = b_i$. We will seek to solve these systems of linear equations.

Example a system of 3 equations in 3 unknowns and its solution is

$$\left\{ \begin{array}{rcrcrcr} 4x_1 & + & x_2 & + & 2x_3 & = & 24 \\ 2x_1 & - & x_2 & - & 2x_3 & = & -6 \\ -x_1 & + & 2x_2 & - & x_3 & = & -4 \end{array} \right. \Longrightarrow \left\{ \begin{array}{l} x_1 = 3 \\ x_2 = 2 \\ x_3 = 5 \end{array} \right.$$

$\square$

These systems can be represented as a matrix equation $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is the $m \times n$ matrix of coefficients, a_{ij}, $\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$ is the n-dimensional column vector of unknowns and $\mathbf{b}$ is a vector of dimension m.

Example cont. In the example above, $\mathbf{A} = \begin{bmatrix} 4 & 1 & 2 \\ 2 & -1 & -2 \\ -1 & 2 & -1 \end{bmatrix}$ and $\mathbf{b} = \begin{bmatrix} 24 \\ -6 \\ -4 \end{bmatrix}$ $\square$

We'll initially look at systems of n equations and n unknowns. Systems with $m < n$ are known as *under-determined* as there are less equations than unknowns while systems with $m > n$ are *over-determined* with more equations than there are unknowns.

When A is non-singular (so A^{-1} exists), the system has a unique solution given by $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$.

Recall that $\mathbf{A}$ is nonsingular if and only if:

(i) inverse matrix $\mathbf{A}^{-1}$ exists; or
(ii) $\det(\mathbf{A}) \neq 0$; or
(iii) $\text{rank}(\mathbf{A}) = m$, or
(iv) $\mathbf{Ax} \neq \mathbf{0}$ for any vector $\mathbf{x} \neq \mathbf{0}$, or
(v) $\text{range}(\mathbf{A}) = \mathbb{R}^m$, or
(vi) $\text{null}(\mathbf{A}) = \{\mathbf{0}\}$.

If $\mathbf{A}$ is singular, the system may have infinitely many solutions or no solutions at all, depending on $\mathbf{b}$.

Example If $\mathbf{A} = \begin{bmatrix} 2 & 3 \\ 4 & 6 \end{bmatrix}$, $\mathbf{Ax} = \mathbf{b}$ has no solution if $\mathbf{b} \notin \text{range}(\mathbf{A})$ or infinitely many solutions when $\mathbf{b} \in \text{range}(\mathbf{A})$. Thus, when $\mathbf{b} = \begin{bmatrix} 4 \\ 7 \end{bmatrix}$ there is no solution, while when $\mathbf{b} = \begin{bmatrix} 4 \\ 8 \end{bmatrix}$, $\mathbf{x} = \begin{bmatrix} \gamma \\ \frac{2}{3}(2-\gamma) \end{bmatrix}$ is a solution for any real γ. $\square$

6 Solving linear equations

In principle, all we need to do to solve the system of equations $\mathbf{Ax} = \mathbf{b}$ is find the inverse of $\mathbf{A}$, $\mathbf{A}^{-1}$. Then $\mathbf{Ax} = \mathbf{b} \implies \mathbf{A}^{-1}\mathbf{Ax} = \mathbf{A}^{-1}\mathbf{b} \implies \mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. In practice, however, things are more complicated. First, $\mathbf{A}$ only has an inverse if it is square (so $m = n$) and $det(A) \neq 0$. In most cases, $m \neq n$ and often even when $m = n$, $det(A) = 0$ is not unusual. Second, supposing that A is indeed square, m and n are often large (10^4 is common, as are much larger values). In these cases, even calculating $det(A)$ is a hugely expensive and complex computational task while finding $\mathbf{A}^{-1}$ is even harder.

We'll initially concentrate on easily solvable systems and look at how we can coerce other systems into a form where they (or some close approximation) too are easily solvable.

6.1 Easily solvable systems 1: Diagonal matrix

All the simple systems we consider here are assumed to be square, so $m = n$. We want to solve $\mathbf{Ax} = \mathbf{b}$.

$\mathbf{A}$ is *diagonal* all entries the off-diagonal are zero. That is $a_{ij} = 0$ when $i \neq j$. So to specify a diagonal matrix, we need only specify the n diagonal elements. We can thus use the simplifying notation, $\mathbf{A} = \text{diag}\{a_1, \ldots, a_n\}$.

When A is diagonal, $x_i = \frac{b_i}{a_i}$ for all $i = 1, \ldots, n$. That is, $\mathbf{A}^{-1} = \text{diag}\{\frac{1}{a_1}, \ldots, \frac{1}{a_n}\}$. Or, to use less compact notation:

$$\mathbf{A} = \begin{bmatrix} a_1 & & \\ & \ddots & \\ & & a_n \end{bmatrix} \Rightarrow \mathbf{A}^{-1} = \begin{bmatrix} \frac{1}{a_1} & & \\ & \ddots & \\ & & \frac{1}{a_n} \end{bmatrix}.$$

6.2 Easily solvable systems 2: Triangular matrix

A matrix is *lower triangular* when all entries above the main diagonal are 0. That is, $\mathbf{A}$ is lower triangular if and only if $a_{ij} = 0$ when $i < j$. Similarly, a matrix is *upper triangular* when all entries above the main diagonal are 0 ($a_{ij} = 0$ for $i > j$). Lower triangular is also called left triangular, and upper called right triangular, for obvious reasons. E.g., a lower triangular matrix:

$$\mathbf{A} = \begin{bmatrix} a_{11} & 0 & \dots & 0 \\ a_{21} & a_{22} & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}.$$

The system $\mathbf{Ax} = \mathbf{b}$ is easy to solve for triangular $\mathbf{A}$ and it does not require that we calculate the inverse of $\mathbf{A}$.

For the lower triangular matrix, the solution is given by

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j \right),$$

so that

$$x_1 = \frac{b_1}{a_{11}};\ x_2 = \frac{b_2 - a_{21}x_1}{a_{22}};\ \dots;\ x_n = \frac{b_n - a_{n1}x_1 - \dots - a_{n-1,n}x_{n-1}}{a_{nn}}.$$

A similar simple formula is available for the upper triangular case, this time working backwards from x_n:

$$x_n = \frac{b_n}{a_{nn}}$$

and

$$x_i = \frac{1}{a_{ii}} \left(b_i - a_{i,i+1}x_{i+1} - \dots - a_{i,n}x_n\right) \text{ for } i = n-1, \dots, 1.$$

The method of Gaussian elimination, or row reduction, which we assume you have seen before transforms the matrix $\mathbf{A}$ into a triangular one to solve the system. This method is reviewed and discussed in Section 7.1

6.3 Easily solvable systems 3: Orthonormal or orthogonal matrix

Matrix $\mathbf{A}$ is *orthogonal* or *orthonormal* if the columns of $\mathbf{A}$ are mutually orthogonal unit vectors.

That is, $\mathbf{A} = [\mathbf{a}_1\ \mathbf{a}_2\ \dots\ \mathbf{a}_n]$ where $\mathbf{a}_i = [a_{i1}\ a_{i2}\ \dots\ a_{in}]^{\mathsf{T}}$ are unit vectors and the set $\{\mathbf{a}_1\ \mathbf{a}_2\ \dots\ \mathbf{a}_n\}$ is mutually orthogonal (so $\mathbf{a}_i \cdot \mathbf{a}_j = 0$ for $i \neq j$).

When $\mathbf{A}$ is orthonormal, $\mathbf{A}^{-1} = \mathbf{A}^T$. This result is true since

$$\mathbf{A}^{\mathsf{T}}\mathbf{A} \equiv \begin{bmatrix} \mathbf{a}_1^{\mathsf{T}} \\ \vdots \\ \mathbf{a}_n^{\mathsf{T}} \end{bmatrix} [\mathbf{a}_1\ \mathbf{a}_2\ \dots\ \mathbf{a}_n] = \mathbf{I}_n \equiv \operatorname{diag}\{1, 1, \dots, 1\}.$$

Also check that $\mathbf{A}\mathbf{A}^{\mathsf{T}} = \mathbf{I}_n$: $\mathbf{A}\mathbf{A}^{\mathsf{T}}\mathbf{A} \equiv \mathbf{A}\underbrace{(\mathbf{A}^{\mathsf{T}}\mathbf{A})}_{\mathbf{I}_n} = \mathbf{A}$ and $\mathbf{A}\mathbf{A}^{\mathsf{T}}\mathbf{A} \equiv (\mathbf{A}\mathbf{A}^{\mathsf{T}})\mathbf{A}$

These properties can be taken as a definition of an orthonomal matrix: $\mathbf{A}^{-1} = \mathbf{A}^T$ if and only if $\mathbf{A}$ is orthonormal.

Thus, if $\mathbf{A}$ is orthonormal, the solution to $\mathbf{Ax} = \mathbf{b}$ is simply $\mathbf{x} = \mathbf{A}^T\mathbf{b}$.

Example: Find the solution to the set of equations

$$\begin{aligned} 0.48x_1 + 0.64x_2 + 0.60x_3 &= 3.56 \\ 0.36x_1 + 0.48x_2 - 0.80x_3 &= -1.08 \\ 0.80x_1 - 0.60x_2 &= -0.40 \end{aligned}$$

or

$$x_1 \overbrace{\begin{bmatrix} 0.48 \\ 0.36 \\ 0.80 \end{bmatrix}}^{\mathbf{a}_1} + x_2 \overbrace{\begin{bmatrix} 0.64 \\ 0.48 \\ -0.60 \end{bmatrix}}^{\mathbf{a}_2} + x_3 \overbrace{\begin{bmatrix} 0.60 \\ -0.80 \\ 0.00 \end{bmatrix}}^{\mathbf{a}_3} = \begin{bmatrix} 3.56 \\ -1.08 \\ -0.40 \end{bmatrix}.$$

So $\mathbf{A} = [\mathbf{a}_1\, \mathbf{a}_2\, \mathbf{a}_3]$.

Solution: By checking that $\mathbf{a}_i \cdot \mathbf{a}_j = 1$ for $i = j = 0$ for $i \neq j$, it is easy to see that $\mathbf{A}$ is orthonormal. So we have the solution

$$\mathbf{x} = \mathbf{A}^{\mathsf{T}}\mathbf{b} = \begin{bmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \mathbf{a}_3^T \end{bmatrix} \mathbf{b}$$

and

$$\begin{aligned} x_1 = \mathbf{a}_1^T\mathbf{b} &= 0.48 \cdot 3.56 - 0.36 \cdot 1.08 - 0.80 \cdot 0.40 \\ &= 1.7088 - 0.3888 - 0.3200 && = 1.0 \\ x_2 = \mathbf{a}_2^T\mathbf{b} &= 0.64 \cdot 3.56 - 0.48 \cdot 1.08 + 0.60 \cdot 0.40 \\ &= 2.2784 - 0.5184 + 0.2400 && = 2.0 \\ x_3 = \mathbf{a}_3^T\mathbf{b} &= 0.60 \cdot 3.56 + 0.80 \cdot 1.08 \\ &= 2.136 + 0.864 && = 3.0. \end{aligned}$$

□

7 Factorising matrices

As we saw in the previous section, matrices with special forms are often much easier to work with than arbitrary matrices. The remainder of this part of the course is focused on how we can manipulate an arbitrary given matrix into a form that is convenient for a stated problem. This is known as *factorising* or *decomposing* matrices.

There are 3 factorisations we will study in various degrees of depth: LU-factorisation, Singular Value Decomposition (SVD) and QR decomposition. A brief summary is given here:

- **Elimination** (LU decomposition): $\mathbf{A} = \mathbf{LU}$
  - Lower triangular matrix ◣ $\times$ ◥ Upper triangular matrix
- **Singular Value Decomposition** (SVD): $\mathbf{A} = \mathbf{UDV}^{\mathsf{T}}$

  - ▥ $\times$ ⧅ diag(singular values) $\times$ ▤ Orthogonal (rows)
  - Orthonormal columns in $\mathbf{U}$ and $\mathbf{V}$:
    the left and right **singular vectors**, respectively
  - Left singular vector: an **eigenvector** of the square $m \times m$ matrix $\mathbf{A}\mathbf{A}^\mathsf{T}$
  - Right singular vector: an **eigenvector** of the square $n \times n$ matrix $\mathbf{A}^\mathsf{T}\mathbf{A}$
  - Singular value: the square root of an eigenvalue of $\mathbf{A}^\mathsf{T}\mathbf{A}$ (or $\mathbf{A}\mathbf{A}^\mathsf{T}$).

- **Orthogonalisation** (QR decomposition): $\mathbf{A} = \mathbf{QR}$

  - Orthogonal matrix (columns) ▥ $\times$ ◥

7.1 LU decomposition via Gaussian elimination

7.1.1 Gaussian elimination to solve systems linear equations (review)

You should be familiar with the process of *Gaussian elimination* (or *row reduction*) in which the equation $\mathbf{Ax} = \mathbf{b}$ (where A is arbitrary) in transformed into the equivalent equation $\mathbf{C}x = \mathbf{d}$ where $\mathbf{C}$ is triangular, making the equation easy to solve. We review the process here.

It is easy to show that *multiplying* both sides of $\mathbf{Ax} = \mathbf{b}$ from the left by any nonsingular matrix $\mathbf{M}$ does not affect the solution. That is $\mathbf{MAx} = \mathbf{Mb}$ has the same solution as $\mathbf{Ax} = \mathbf{b}$, since

$$\mathbf{MAx} = \mathbf{Mb} \Rightarrow \mathbf{x} = (\mathbf{MA})^{-1}\,\mathbf{Mb} = \mathbf{A}^{-1}\mathbf{M}^{-1}\mathbf{Mb} = \mathbf{A}^{-1}\mathbf{b}.$$

We know from the above result that we can multiple both sides by a series of *elementary matrices* which perform various *row operations* on $\mathbf{A}$: the three types of operation are row swapping, row multiplication and adding some multiple of one row to another row. Repeated application of these three operations (that is, repeated multiplication by elementary matrices) to both sides of the equation transforms it to $\mathbf{C}x = \mathbf{d}$ where $\mathbf{C} = \mathbf{M}_1 \ldots \mathbf{M}_k\mathbf{A}$ is in upper triangular form (so the only non-zero elements of $\mathbf{C}$ are on or above the diagonal) and $\mathbf{d} = \mathbf{M}_1 \ldots \mathbf{M}_k\mathbf{b}$.

Example: Use Gaussian elimination to solve the system of equations $\mathbf{Ax} = \mathbf{b}$ where

$$\mathbf{A} = \begin{bmatrix} 3 & 2 & 1 & 2 \\ 6 & 6 & 3 & 5 \\ 3 & 0 & 3 & 5 \\ 9 & 2 & 7 & 8 \end{bmatrix} \text{ and } \mathbf{b} = \begin{bmatrix} 4 \\ 5 \\ 5 \\ 10 \end{bmatrix}$$

Solution:

$$\overbrace{\begin{bmatrix} 3 & 2 & 1 & 2 \\ 6 & 6 & 3 & 5 \\ 3 & 0 & 3 & 5 \\ 9 & 2 & 7 & 8 \end{bmatrix}}^{\mathbf{A}} \overbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}^{\mathbf{x}} = \overbrace{\begin{bmatrix} 4 \\ 5 \\ 5 \\ 10 \end{bmatrix}}^{\mathbf{b}}$$

$$\Rightarrow \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ -3 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3 & 2 & 1 & 2 \\ 6 & 6 & 3 & 5 \\ 3 & 0 & 3 & 5 \\ 9 & 2 & 7 & 8 \end{bmatrix}}_{\text{Eliminating the first column: } \mathbf{M}_1\mathbf{A}} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_{\mathbf{x}} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ -3 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 \\ 5 \\ 5 \\ 10 \end{bmatrix}}_{\mathbf{M}_1\mathbf{b}}$$

$$\Rightarrow \begin{bmatrix} 3 & 2 & 1 & 2 \\ 0 & 2 & 1 & 1 \\ 0 & -2 & 2 & 3 \\ 0 & -4 & 4 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 4 \\ -3 \\ 1 \\ -2 \end{bmatrix}$$

$$\Rightarrow \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3 & 2 & 1 & 2 \\ 0 & 2 & 1 & 1 \\ 0 & -2 & 2 & 3 \\ 0 & -4 & 4 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_{\mathbf{M}_2\mathbf{M}_1\mathbf{A}\mathbf{x}} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 \\ -3 \\ 1 \\ -2 \end{bmatrix}}_{\mathbf{M}_2\mathbf{M}_1\mathbf{b}}$$

$$\Rightarrow \begin{bmatrix} 3 & 2 & 1 & 2 \\ 0 & 2 & 1 & 1 \\ 0 & 0 & 3 & 4 \\ 0 & 0 & 6 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 4 \\ -3 \\ -2 \\ -8 \end{bmatrix}$$

$$\Rightarrow \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -2 & 1 \end{bmatrix} \begin{bmatrix} 3 & 2 & 1 & 2 \\ 0 & 2 & 1 & 1 \\ 0 & 0 & 3 & 4 \\ 0 & 0 & 6 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_{\mathbf{M}_3\mathbf{M}_2\mathbf{M}_1\mathbf{A}\mathbf{x}} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -2 & 1 \end{bmatrix} \begin{bmatrix} 4 \\ -3 \\ -2 \\ -8 \end{bmatrix}}_{\mathbf{M}_3\mathbf{M}_2\mathbf{M}_1\mathbf{b}}$$

$$\Rightarrow \begin{bmatrix} 3 & 2 & 1 & 2 \\ 0 & 2 & 1 & 1 \\ 0 & 0 & 3 & 4 \\ 0 & 0 & 0 & -4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 4 \\ -3 \\ -2 \\ -4 \end{bmatrix}$$

It is easy to see that the solution to this row reduced matrix equation is

$$\begin{aligned} x_4 &= \tfrac{-4}{-4} &&= 1 \\ x_3 &= \tfrac{1}{3}(-2 - 4 \cdot 1) &&= -2 \\ x_2 &= \tfrac{1}{2}(-3 - 1 \cdot (-2) - 1 \cdot 1) &&= -1 \\ x_1 &= \tfrac{1}{3}(4 - 2 \cdot (-1) - 1 \cdot (-2) - 2 \cdot 1) &&= 2 \end{aligned}$$

$\square$

7.1.2 Gaussian elimination as LU decomposition

It turns out that Gaussian elimination can be viewed a LU decomposition in which a matrix $\mathbf{A}$ is written as the product of a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$ so that $\mathbf{A} = \mathbf{LU}$. Recall that the Gaussian elimination process starts with an arbitrary square matrix $\mathbf{A}$, multiplies it by a series of elementary vectors, $\mathbf{M}_1 \dots \mathbf{M}_k$ to get $\mathbf{U} = \mathbf{M}_1 \dots \mathbf{M}_k \mathbf{A}$ where $\mathbf{U}$ is upper triangular (we called it $\mathbf{C}$ in the earlier discussion).

Now (check that you understand the following statements), each of the elementary matrices is lower triangular (so long as there are no row swapping operations) and the inverse of lower triangular matrices are lower triangular too, so each of $\mathbf{M}_i^{-1}$, for $i = 1, \dots, k$ is lower triangular. Finally, the product of lower triangular matrices is also lower triangular so

$$\mathbf{U} = \mathbf{M}_1 \dots \mathbf{M}_k \mathbf{A} \implies \mathbf{LU} = \mathbf{A},$$

where $\mathbf{L} = (\mathbf{M}_1 \dots \mathbf{M}_k)^{-1} = \mathbf{M}_k^{-1} \dots \mathbf{M}_1^{-1}$. If row permutations (row swaps) are needed in the Gaussian elimination process, we can't find an LU decomposition for $\mathbf{A}$ but can find an LU decomposition for the permuted matrix $\mathbf{PA}$, where $\mathbf{P}$ describes the necessary permutations. Obviously, the permuted system has the same solution as the unpermuted system.

The computational complexity of solving a system of n equations in n unknown using Gaussian elimination is $O(n^3)$. It is typically a stable algorithm, though potential for instability arises when a leading non-zero entry is very small (as we divide through by this entry). Reordering of the rows before the start of the row reduction process so that the largest leading non-zero elements are selected first can avoid this cause of instability. This technique is known as pivoting.

We won't look further at LU decompositions but will spend considerable time looking first at singular value decomposition and its uses and, later, when we consider the Least Squares framework, QR decompositions and its applications.

7.2 Eigenvalues and eigenvectors of real symmetric matrices[1]

Result: in elementary linear algebra courses, it is shown that a real symmetric (so square) matrix $\mathbf{A}$ always has real eigenvalues and the eigenvectors of such a matrix may always be chosen to form an orthonormal set of size n.

Denote the eigenvectors of $\mathbf{A}$ by $\mathbf{u}_i$ with corresponding eigenvalue λ_i, so that

$$\mathbf{A}\mathbf{u}_i = \mathbf{u}_i \lambda_i \tag{3}$$

[1]This section is taken, with few modifications, from notes written by Sze Tan for Physics 707: Inverse Problems.

(note that I purposely write the right hand side this order. You can consider it as matrix multiplication of a $n \times 1$ matrix with a 1×1 matrix. Of course, the result is the same as standard scalar multiplication of a matrix $\lambda_i \mathbf{u}_i$.)

Write the column vectors $\mathbf{u}_i, \ldots, \mathbf{u}_n$ as the columns of a square matrix:

$$\mathbf{U} = \begin{bmatrix} \vdots & \vdots & & \vdots \\ \mathbf{u}_1 & \mathbf{u}_2 & \ldots & \mathbf{u}_n \\ \vdots & \vdots & & \vdots \end{bmatrix}$$

Now the relations described by Equation 3 can be written

$$\mathbf{AU} = \begin{bmatrix} \vdots & \vdots & & \vdots \\ \mathbf{Au}_1 & \mathbf{Au}_2 & \ldots & \mathbf{Au}_n \\ \vdots & \vdots & & \vdots \end{bmatrix} = \begin{bmatrix} \vdots & \vdots & & \vdots \\ \lambda_1\mathbf{u}_1 & \lambda_2\mathbf{u}_2 & \ldots & \lambda_n\mathbf{u}_n \\ \vdots & \vdots & & \vdots \end{bmatrix}$$

$$= \begin{bmatrix} \vdots & \vdots & & \vdots \\ \mathbf{u}_1 & \mathbf{u}_2 & \ldots & \mathbf{u}_n \\ \vdots & \vdots & & \vdots \end{bmatrix} \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{bmatrix} = \mathbf{UD}$$

where $\mathbf{D}$ is the diagonal matrix with eigenvalues on the diagonal, $\mathbf{D} = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$. Since $\mathbf{U}$ is orthogonal (and assuming its columns are normalised), it is invertible with $\mathbf{U}^{-1} = \mathbf{U}^T$ so

$$\mathbf{A} = \mathbf{UDU}^T.$$

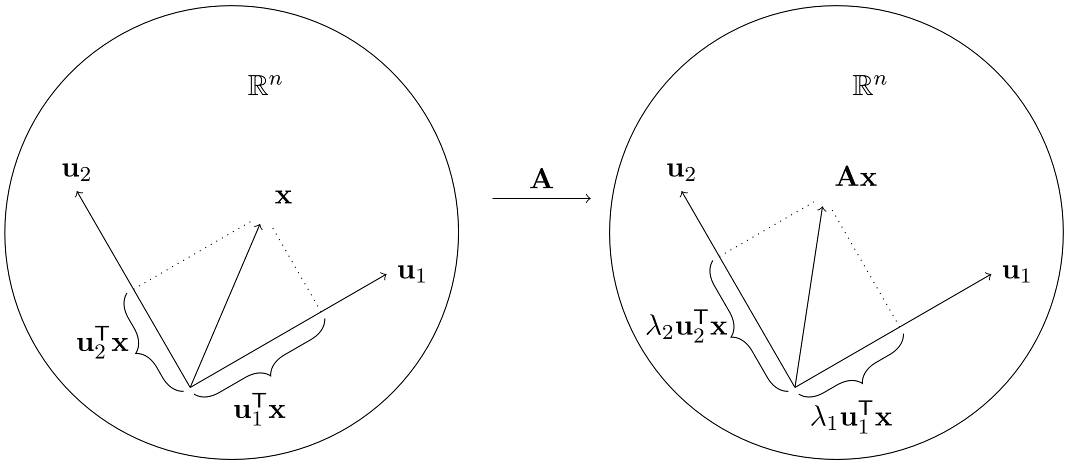
This can be rewritten as

$$\mathbf{A} = \sum_{k=1}^{n} \lambda_k \mathbf{u}_k \mathbf{u}_k^T,$$

a decomposition that we have seen before that is worth considering further. First, you can check that the decomposition is actually correct by showing that the multiplying the eigenvectors $\mathbf{u}_i$ by $\mathbf{A}$ and $\sum_{k=1}^{n} \lambda_k \mathbf{u}_k \mathbf{u}_k^T$ produces equivalent results.

Second, it means that the action of multiplying an arbitrary n-vector $\mathbf{x}$ by the real symmetric matrix $\mathbf{A}$, so $\mathbf{Ax} = (\sum_{k=1}^{n} \lambda_k \mathbf{u}_k \mathbf{u}_k^T)\mathbf{x} = \sum_{k=1}^{n} \mathbf{u}_k \lambda_k \mathbf{u}_k^T \mathbf{x}$ can be understood as comprising three steps:

1. It resolves the input vector along each of the eigenvectors $\mathbf{u}_k$, the component of the input vector along the kth eigenvector being given by $\mathbf{u}_k^T\mathbf{x}$,
2. The amount along the kth eigenvector is multiplied by the eigenvalue λ_k,
3. The product tells us how much of the kth eigenvector $\mathbf{u}_k$ is present in the product $\mathbf{Ax}$.



$\mathbb{R}^n$
$\mathbf{u}_2$
$\mathbf{x}$
$\mathbf{u}_1$
$\mathbf{u}_2^\mathsf{T}\mathbf{x}$
$\mathbf{u}_1^\mathsf{T}\mathbf{x}$
$\mathbf{A}$
$\mathbb{R}^n$
$\mathbf{u}_2$
$\mathbf{Ax}$
$\mathbf{u}_1$
$\lambda_2\mathbf{u}_2^\mathsf{T}\mathbf{x}$
$\lambda_1\mathbf{u}_1^\mathsf{T}\mathbf{x}$

Figure 3: Effect of a real symmetric matrix $\mathbf{A}$ of size n on a vector $\mathbf{x}$. Only two of the orthogonal eigenvectors are shown.

A schematic diagram of this process is shown in Figure 3.

Note that this is a special case of *diagonalisation*. A matrix $\mathbf{A}$ is diagonalisable if $\mathbf{A} = \mathbf{PDP}^{-1}$ (or, equivalently, $\mathbf{P}^{-1}\mathbf{AP} = \mathbf{D}$) for some diagonal matrix $\mathbf{D}$ and some matrix $\mathbf{P}$. In the case discussed above, where $\mathbf{A}$ is real and symmetric, $\mathbf{A}$ is diagonalisable with $\mathbf{P} = \mathbf{U}$ and $\mathbf{U}^{-1} = \mathbf{U}^\mathrm{T}$.

8 Singular Value Decomposition (SVD)

Singular Value Decomposition is a method of factorising any ordinary rectangular $m \times n$ matrix. It is most frequently applied to problems where $m \geq n$ (more equations than unknowns).

It has applications in signal processing, pattern recognition, and statistics where it is used for least squares data fitting, regularised inverse problems, finding pseudoinverses and performing principal component analysis (PCA). The application areas are many and varied but include computational tomography, seismology, weather forecast, image compression, image denoising, genetic analyses and more.

It is particularly useful when a given set of linear equations is singular or very close to singular in which case conventional solutions (e.g. by LU decomposition) are either not available or produce senseless results (due to the problems being ill-posed). In these cases, SVD can diagnose and, in some cases, solve the problem giving an useful numerical answer (though not necessarily the expected one!).

8.1 Overview of an SVD

SVD represents an ordinary $m \times n$ matrix $\mathbf{A}$ as $\mathbf{A} = \mathbf{UDV}^\mathsf{T}$ where:

U : an $m \times m$ column-orthogonal matrix; its m columns are the m eigenvectors $\mathbf{u}$ of the $m \times m$ matrix $\mathbf{AA}^\mathsf{T}$. The vectors $\{\mathbf{u}\}$ are known as the *left singular vectors* of $\mathbf{A}$.

V : an $n \times n$ orthogonal matrix; its n columns are the eigenvectors $\mathbf{v}$ of the $n \times n$ matrix $\mathbf{A}^\mathsf{T}\mathbf{A}$. The vectors $\{\mathbf{v}\}$ are known as the *right singular vectors* of $\mathbf{A}$.

D : an $m \times n$ matrix whose only non-zero elements are the first r entries on the diagonal where r is the rank of $\mathbf{A}$ and $d_{kk} = \sigma_k = \sqrt{\lambda_k}$ where λ_k is the eigenvalue associated with $\mathbf{v}_k$.

The *singular values*, σ_k are ordered so that $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r > 0$.
Since $\mathbf{A} = \mathbf{UDV}^\mathsf{T}$, we can write

$$\begin{aligned} \mathbf{A} &= \sum_{k=1}^{r} \sigma_k \mathbf{u}_k \mathbf{v}_k^T \\ &= \sigma_1 \mathbf{u}_1 \mathbf{v}_1^\mathsf{T} + \sigma_2 \mathbf{u}_2 \mathbf{v}_2^\mathsf{T} + \ldots + \sigma_r \mathbf{u}_r \mathbf{v}_r^\mathsf{T}. \end{aligned} \tag{4}$$

This representation suggests the approximation of $\mathbf{A}$ by the truncated series,

$$\widehat{\mathbf{A}}_\rho = \sum_{k=1}^{\rho} \sigma_k \mathbf{u}_k \mathbf{v}_k^T \text{ for } \rho < r.$$

Notice that when $m > n$ (that is, the problem is over-determined), there are at most n non-zero singular values. In this case, we can truncate the matrix $\mathbf{U}$ to be $m \times n$ and the matrix $\mathbf{D}$ to be a $n \times n$ diagonal matrix. This leaves the sum in Equation 4 unaltered as the rows or columns that are removed contribute nothing to that sum. In the following example, we employ this strategy.

Example: Find the SVD of the matrix $\mathbf{A}$ where

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

Solution: First find the eigenvectors and eigenvalues of $\mathbf{AA}^\mathsf{T}$ and $\mathbf{A}^\mathsf{T}\mathbf{A}$. Since $\mathbf{A}$ is 3×2, we need only find the top two eigenvalues and eigenvectors of each of these matrices.

$$\mathbf{A}^\mathsf{T}\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

which has eigenvalues $\lambda_1 = 3$ and $\lambda_2 = 1$. The associated eigenvectors are, respectively,

$$\mathbf{v}_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \text{ and } \mathbf{v}_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} -1 \\ 1 \end{bmatrix}.$$

Notice that the eigenvectors have been normalised.
Similarly,

$$\mathbf{A}\mathbf{A}^\mathsf{T} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

The top two eigenvalues are $\mu_1 = 3$ and $\mu_2 = 1$. Notice that these are the same as the top two eigenvalues of $\mathbf{A}^\mathsf{T}\mathbf{A}$. The associated eigenvectors are

$$\mathbf{u}_1 = \frac{1}{\sqrt{6}} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \text{ and } \mathbf{u}_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}.$$

The singular values are given by $\sigma_i = \sqrt{\lambda_i}$, so $\sigma_1 = \sqrt{3}$ and $\sigma_2 = 1$.
We can thus write $\mathbf{A}$ as

$$\begin{aligned} \mathbf{A} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 1 & 0 \end{bmatrix} &= \underbrace{[\mathbf{u}_1\ \mathbf{u}_2]}_{\mathbf{U}} \underbrace{\operatorname{diag}(\sigma_1, \sigma_2)}_{\mathbf{D}} \underbrace{\begin{bmatrix} \mathbf{v}_1^\mathsf{T} \\ \mathbf{v}_2^\mathsf{T} \end{bmatrix}}_{\mathbf{V}^\mathsf{T}} \\ &= \underbrace{\begin{bmatrix} 1/\sqrt{6} & 1/\sqrt{2} \\ 2/\sqrt{6} & 0 \\ 1/\sqrt{6} & -1/\sqrt{2} \end{bmatrix}}_{\mathbf{U}} \underbrace{\begin{bmatrix} \sqrt{3} & 0 \\ 0 & 1 \end{bmatrix}}_{\mathbf{D}} \underbrace{\begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ -1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix}}_{\mathbf{V}^\mathsf{T}}. \end{aligned}$$

The matrix approximation $\widehat{\mathbf{A}}_1$ is calculated as follows:

$$\begin{aligned} \widehat{\mathbf{A}}_1 &= \sigma_1 \mathbf{u}_1 \mathbf{v}_1^\mathsf{T} \\ &= \sqrt{3} \cdot \tfrac{1}{\sqrt{6}} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \cdot \tfrac{1}{\sqrt{2}} [1\ \ 1] = \tfrac{1}{2} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} [1\ \ 1] = \begin{bmatrix} 0.5 & 0.5 \\ 1 & 1 \\ 0.5 & 0.5 \end{bmatrix} \end{aligned}$$

while the approximation can be extended to $\widehat{\mathbf{A}}_2(=\mathbf{A})$ by

$$\begin{aligned}
\widehat{\mathbf{A}}_2 &= \widehat{\mathbf{A}}_1 + \sigma_2 \mathbf{u}_2 \mathbf{v}_2^{\mathrm{T}} \\
&\equiv \begin{bmatrix} 0.5 & 0.5 \\ 1 & 1 \\ 0.5 & 0.5 \end{bmatrix} + 1 \cdot \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \frac{1}{\sqrt{2}} [-1 \;\; 1] \\
&\equiv \begin{bmatrix} 0.5 & 0.5 \\ 1 & 1 \\ 0.5 & 0.5 \end{bmatrix} + \begin{bmatrix} -0.5 & 0.5 \\ 0 & 0 \\ 0.5 & -0.5 \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 1 & 0 \end{bmatrix}}_{\mathbf{A}}.
\end{aligned}$$

8.2 How does this all work?[2]

Recall $\mathbf{A}$ is $m \times n$ so that $\mathbf{A}^T\mathbf{A}$ is $n \times n$ and $\mathbf{A}\mathbf{A}^T$ is $m \times m$. Both of these product matrices are square and symmetric.

So, by the result we saw earlier, $\mathbf{A}^T\mathbf{A}$ has n real eigenvalues and a set of n orthonormal eigenvectors (similarly for $\mathbf{A}\mathbf{A}^T$ which has m of them).

Let $\mathbf{v}_i$ be the eigenvectors of $\mathbf{A}\mathbf{A}^T$ and λ_i be the corresponding eigenvectors and order them so that $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n \geq 0$. (It can be shown that all eigenvales here are ≥ 0.)

Similarly, let $\mathbf{u}_i$ be the eigenvectors of $\mathbf{A}^T\mathbf{A}$ and μ_i be the corresponding eigenvectors and order them so that $\mu_1 \geq \mu_2 \geq \ldots \geq \mu_m \geq 0$.

It turns out that the non-zero eigenvalues of $\mathbf{A}\mathbf{A}^T$ are exactly the same as the non-zero eigenvalues of $\mathbf{A}^T\mathbf{A}$. Suppose there are r such non-zero eigenvalues, so that $\lambda_{r+1} = \ldots = \lambda_n = 0$ and $\mu_{r+1} = \ldots = \mu_m = 0$.

r is called the *rank* of $\mathbf{A}$ (and of $\mathbf{A}^T$). Clearly, $r \leq m$ and $r \leq n$.

Now, for $k = 1, \ldots, r$, it can be shown that we have

$$\mathbf{A}\mathbf{v}_k = \sigma_k \mathbf{u}_k \text{ and } \mathbf{A}^T\mathbf{u}_k = \sigma_k \mathbf{v}_k$$

where $\sigma_k = \sqrt{\lambda}_k = \sqrt{\mu}_k$. And, also that, for $k > r$,

$$\mathbf{A}\mathbf{v}_k = 0 \text{ and } \mathbf{A}^T\mathbf{u}_k = 0.$$

The equations $\mathbf{A}\mathbf{v}_k = \sigma_k \mathbf{u}_k$ for $k \leq r$ together with $\mathbf{A}\mathbf{v}_k = 0$ for $k > r$ tell us how $\mathbf{A}$ acts on the orthonormal set of vectors $\{\mathbf{v}_k\}$. Since this set is a basis for $\mathbb{R}^n$, the equations give a complete description of the action of $\mathbf{A}$, so that we can write

$$\mathbf{A} = \sum_{k=1}^{r} \sigma_k \mathbf{u}_k \mathbf{v}_k^{\mathrm{T}}. \tag{5}$$

[2]This section is taken, and condensed, from notes written by Sze Tan for Physics 707: Inverse Problems.

A similar argument shows that

$$\mathbf{A}^{\mathrm{T}} = \sum_{k=1}^{r} \sigma_k \mathbf{v}_k \mathbf{u}_k^{\mathrm{T}}.$$

The orthonormal vectors $\{\mathbf{v}_k\}$ are known as the *right singular vectors*, the vectors $\{\mathbf{u}_k\}$ are known as the *left singular vectors*, and the scalars $\{\sigma_k\}$ are called the *singular values* of the matrix $\mathbf{A}$.

The singular value decomposition allows us to understand the action of $\mathbf{A}$ on a vector $\mathbf{x}$ as

$$\mathbf{A}\mathbf{x} = \sum_{k=1}^{r} \mathbf{u}_k \sigma_k (\mathbf{v}_k^{\mathrm{T}} \mathbf{x}).$$

which can be interpreted as having three stages:

1. It resolves the input vector along each of the right singular vectors $\mathbf{v}_k$, the component of the input vector along the kth singular vector being given by $\mathbf{v}_k^T \mathbf{x}$,
2. The amount along the kth direction is multiplied by the singular value σ_k,
3. The product tells us how much of the kth left singular vector $\mathbf{u}_k$ is present in the product $\mathbf{A}\mathbf{x}$.

This is illustrated in Figure 4.

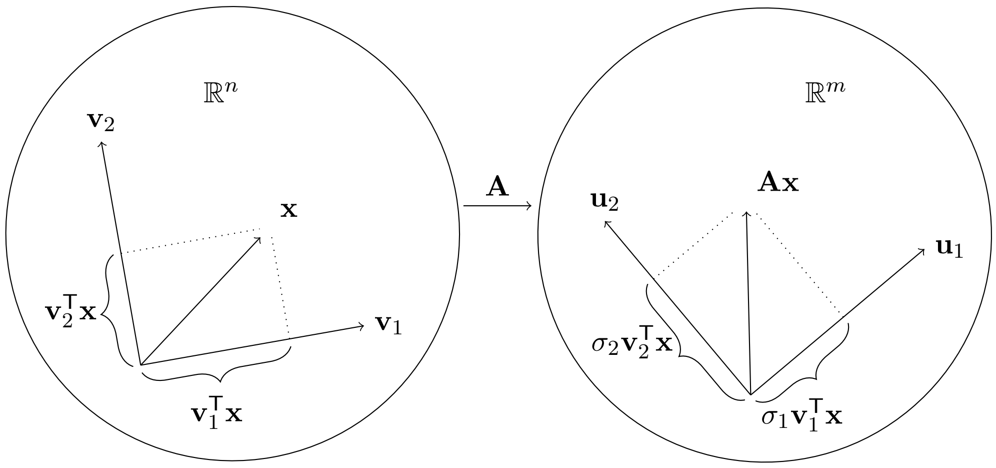
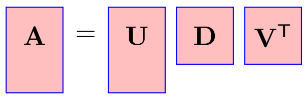

$\mathbb{R}^n$
$\mathbf{v}_2$
$\mathbf{x}$
$\mathbf{v}_1$
$\mathbf{v}_2^{\mathsf{T}}\mathbf{x}$
$\mathbf{v}_1^{\mathsf{T}}\mathbf{x}$
$\mathbf{A}$
$\mathbb{R}^m$
$\mathbf{A}\mathbf{x}$
$\mathbf{u}_2$
$\mathbf{u}_1$
$\sigma_2\mathbf{v}_2^{\mathsf{T}}\mathbf{x}$
$\sigma_1\mathbf{v}_1^{\mathsf{T}}\mathbf{x}$

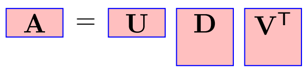
Figure 4: Effect of a rectangular matrix $\mathbf{A}$ of size $m \times n$ on a vector $\mathbf{x}$. Only two of the orthogonal eigenvectors are shown.

8.3 Structure of SVD

In the overdetermined case, in which $m > n$, so that we have more equations than unknowns, we have the following structure:



In the underdetermined case, in which $m < n$, so that we have fewer equations than unknowns, we have the following structure:



Note that, in the overdetermined case, we truncate $\mathbf{U}$ and $\mathbf{D}$ since there are at most $r \leq n < m$ non-zero singular values of $\mathbf{A}$ we can omit the $\mathbf{u}_i$ that contribute nothing to matrix product.

The matrix $\mathbf{V}$ is orthonormal, so $\mathbf{V}\mathbf{V}^T = \mathbf{V}^T\mathbf{V} = \mathbf{I}_n$. $\mathbf{U}$ is orthonormal when $m \geq n$, but if $m < n$, the singular values $\sigma_j = 0$ for $j = m+1, \ldots, n$ and the corresponding columns of $\mathbf{U}$ are also 0 so that $\mathbf{U}\mathbf{U}^T = \text{diag}(1, \ldots, 1, 0, \ldots, 0)$ where only the first m elements of the diagonal are 1 and the elements from $m+1$ to n are zero.

Note that the SVD of matrix $\mathbf{A}$ is only unique up to permutations of the columns/rows. For this reason, we insist that the singular values and corresponding singular vectors are arranged so that the singular values are in descending order $\sigma_1 \geq \sigma_2 \geq \ldots$. Even then, some of the σ_i's may have the same value so columns of $\mathbf{U}$ and $\mathbf{V}$ could be permuted. Aside from these possible permuations, the representation is unique. Be aware when calculating the SVD with various software that the you may need to enforce this canonical representation.

8.4 Condition number of a matrix

The concept of a condition number was introduced in Section 2. This concept can be applied to a matrices and is useful, for example, when considering solutions to the equation $\mathbf{Ax} = \mathbf{b}$. Solutions to this equation will change greatly with small changes in $\mathbf{b}$ when $\mathbf{A}$ has a large condition number, while the small changes in $\mathbf{b}$ will lead to only small changes in the solution when the matrix has a small condition number. We can define the condition number of a matrix as the maximum of the ratio of the relative error in $\mathbf{x}$ divided by the relative error in $\mathbf{b}$, where the maximum is taken over all possible $\mathbf{x}$ and $\mathbf{b}$.

To give a full description of how to derive the condition number of $\mathbf{A}$, we would have to introduce matrix norms which we do not have time to do here. Instead, we simply present the result here that the condition number of $\mathbf{A}$ can be defined as the ratio of the largest to the smallest non-zero singular values:

$$cond(\mathbf{A}) = \frac{\sigma_{\max}}{\sigma_{\min}}.$$

If the smallest singular value of $\mathbf{A}$ is 0, $\mathbf{A}$ is singular (has no inverse) but the condition number of $\mathbf{A}$ is still defined.

The condition number of $\mathbf{A}$ is considered to be large, and the matrix is *ill-conditioned*, if roughly $\log(cond(\mathbf{A})) \geq k$ where k is the number of digits of precision in the matrix entries.

Example: Find the condition number of the matrix $\mathbf{A} = \begin{bmatrix} 2 & -3 \\ 1 & -1 \end{bmatrix}$

Solution: Using a matrix algebra package, find the singular values of $\mathbf{A}$ to be 3.864 and 0.259, so $cond(\mathbf{A}) = \frac{3.864}{0.259} \approx 14.9$. $\square$

Example: The singular values of the matrix $\mathbf{A} = \begin{bmatrix} 1.2969 & 0.8648 \\ 0.2161 & 0.1441 \end{bmatrix}$ are approximately 1.58 and 6.33×10^{-9} so the condition number is about 2.5×10^{8}. This very large condition number means that $\mathbf{A}$ is an ill-conditioned matrix.

Ill-conditioning means that standard approaches to solving linear systems can be very unstable. For example, consider the linear system $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{b} = \begin{bmatrix} 0.8642 \\ 0.1140 \end{bmatrix}$. This has the exact solution $\mathbf{x} = \begin{bmatrix} 2 \\ -2 \end{bmatrix}$.

But standard matrix software (`linsolve` in Matlab), gives the solution as $\begin{bmatrix} 2.59 \\ -3.89 \end{bmatrix} \times 10^{6}$ which is radically wrong!

It also means that if some number in the system is measured slightly differently, the results we get can change enormously. For example, if the measurement vector $\mathbf{b}$ is just slightly different, say $\mathbf{b} = \begin{bmatrix} 0.86419999 \\ 0.11400001 \end{bmatrix}$, then the exact solution is now close to $\mathbf{x} = \begin{bmatrix} 0.9911 \\ -0.4870 \end{bmatrix}$ which represents an enormous change in the solution relative to the small change in the original system. $\square$

We'll see in a later section on pseudo-inverses how the SVD can be used to solve the linear system $\mathbf{Ax} = \mathbf{b}$.

8.4.1 Image compression

See slides and assignment 1.

8.4.2 Gene expression

Abstract from Alter et al, 2000, *Singular value decomposition for genome-wide expression data processing and modeling*, `http://www.pnas.org/content/97/18/10101.full`: We describe the use of singular value decomposition in transforming genome-wide expression data from genes $\times$ arrays space to reduced diagonalized eigengenes $\times$ eigenarrays space, where the eigengenes (or eigenarrays) are unique orthonormal superpositions of the genes (or arrays). Normalizing the data by filtering out the eigengenes (and eigenarrays) that are inferred to represent noise or experimental artifacts enables meaningful comparison of the expression of different genes across different arrays in different experiments. Sorting the data according to the eigengenes and eigenarrays gives a global picture of the dynamics of gene expression, in which individual genes and arrays appear to be classified into groups of similar regulation and function, or similar cellular state and biological phenotype, respectively. After normalization and sorting, the significant eigengenes and eigenarrays can be associated with observed genome-wide effects of regulators, or with measured samples, in which these regulators are overactive or underactive, respectively.

8.5 Principal Components Analysis (PCA)

PCA is a common technique for identifying patterns in high-dimensional data. It transforms the original correlated measurements into uncorrelated measurements. One of the main uses of PCA is as a dimension reduction tool, in which only the directions in which the data varies the most are considered. This can lead to enormous simplifications of the data and provide insights for a wide variety of data. PCA is alternatively known as the Karhunen-Loéve transform (KLT), the Hotelling transform or proper orthogonal decomposition (POD)

These new coordinate axes (along which the data varies the most) are are known as *principal components* and are, by construction, orthogonal.

A useful visualisation tool to aid your understanding of PCA is at http://setosa.io/ev/principal-component-analysis/.

Suppose we have a $m \times n$ matrix of measurement data A. For example, n trials where m properties were measured in each trial. Then, if $\mathbf{a}_i$ are the measurements from the ith trial,

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1 & \mathbf{a}_2 & \dots & \mathbf{a}_n \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}.$$

For example, $\mathbf{A}$ could be $n = 100$ observations of the position of an object measured in $m = 3$ dimensions.

In the following, assume that the rows of $\mathbf{A}$ have been centred, so that the mean of each row is 0 (each rows of $\mathbf{A}$ corresponds to a dimension in the original data). If this is not already the case, it can be achieved by subtracting the mean of each row of $\mathbf{A}$ from each element of that row. That is, set element

$$a_{ij} = a_{ij} - \sum_{j=1}^{n} a_{ij}/n$$

to centre the rows of $\mathbf{A}$. This is a critical assumption and allows us to concentrate on the variance.

Each observation $\mathbf{A}$ is just the m-vector $\mathbf{a}_i$. The idea of PCA is to chose a new basis $\mathbf{u}_1, \ldots, \mathbf{u}_k$ to express the data points (the $\mathbf{a}_i$'s) so that the variance of the measurements is greatest in the direction of $\mathbf{u}_1$, the next greatest variance is in the direction of $\mathbf{u}_2$ and so on, down to $\mathbf{u}_k$. Ideally, $k < m$.

Define the *covariance matrix* of $\mathbf{A}$ by

$$\mathbf{\Sigma} = \frac{1}{n-1}\mathbf{A}\mathbf{A}^T.$$

Then $\mathbf{\Sigma}$ is an $m \times m$ matrix where the diagonal terms of $\mathbf{\Sigma}$ are the variance of the ith dimension of the measurement, while the off-diagonal terms of $\mathbf{\Sigma}$ are the covariances between different measurements.

It turns out that the best basis to choose are the k eigenvectors of $\mathbf{\Sigma}$ corresponding its k largest eigenvalues. These are known as the *principal components* of $\mathbf{A}$. Call them $\mathbf{u}_1, \ldots, \mathbf{u}_k$ and form the matrix

$$\mathbf{U}_k = [\mathbf{u}_1, \ldots, \mathbf{u}_k]$$

From results we have seen earlier about symmetric matrices, this an orthogonal matrix. We include the subscript k as we may decide to truncate this matrix by including only the eigenvectors corresponding to the largest eigenvalues. That is, if $\mathbf{\Sigma}$ has K eigenvectors and associated eigenvalues, and the largest k eigenvalues are substantially larger than the remaining $K - k$, it is reasonable to form $\mathbf{U}_k$ containing only the most significant k eigenvectors.

We can now represent the original measurements, $\mathbf{A}$, in this this new co-ordinate system. The amount of measurement vector $\mathbf{a}_i$ in direction $\mathbf{u}_j$ is given by $\mathbf{u}_j^\top \mathbf{a}_i$: this is the jth coordinate of $\mathbf{a}_i$ in this new coordinate system. So if we consider just the two dimension space defined by the top two principal components, $\mathbf{a}_i$ has coordinates

$$\mathbf{U}_2^\top \mathbf{a}_i = \begin{bmatrix} \mathbf{u}_1^\top \\ \mathbf{u}_2^\top \end{bmatrix} \mathbf{a}_i = \begin{bmatrix} \mathbf{u}_1^\top \mathbf{a}_i \\ \mathbf{u}_2^\top \mathbf{a}_i \end{bmatrix}.$$

We usually consider this space independently of how it relates to the original m-dimensional space but we can consider it as embedded in the original space. To find the coordinates of a point in this embedded space, define the projection matrix, $\mathbf{P}_k$ by

$$\mathbf{P}_k = \mathbf{U}_k \mathbf{U}_k^\top = \begin{bmatrix} \mathbf{u}_1 & \mathbf{u}_2 & \ldots & \mathbf{u}_k \end{bmatrix} \begin{bmatrix} \mathbf{u}_1^\top \\ \mathbf{u}_2^\top \\ \vdots \\ \mathbf{u}_k^\top \end{bmatrix}$$

so that each measurement vector $\mathbf{a}_i$ is projected via $\mathbf{P}_k \mathbf{a}_i$.

One interpretation of PCA is that the projection $\mathbf{P}_k$ is chosen to minimise the projection error $\sum_{j=1}^{n} \|\mathbf{a}_j - \mathbf{P}_k \mathbf{a}_j\|^2$

Example: Find the principal components of the data matrix $\mathbf{A}$ where

$$\mathbf{A} = \begin{bmatrix} -4 & 3 & -5 & 18 & 6 & -5 \\ 2 & 6 & -2 & 10 & 1 & -1 \\ 7 & 11 & 3 & 6 & 9 & 3 \end{bmatrix}.$$

Find the amount of the first principal component in the first measurement vector of $\mathbf{A}$ (that is, the first column), and calculate the projection matrix for projecting $\mathbf{A}$ onto the first two principal components.

Solution: First, centre the rows of $\mathbf{A}$ so that each row has mean zero. Call this centred matrix $\mathbf{B}$.

$$\mathbf{B} = \begin{bmatrix} -6.1667 & 0.8333 & -7.1667 & 15.8333 & 3.8333 & -7.1667 \\ -0.6667 & 3.3333 & -4.6667 & 7.3333 & -1.6667 & -3.6667 \\ 0.5 & 4.5 & -3.5 & -0.5 & 2.5 & -3.5 \end{bmatrix}.$$

Now form the covariance matrix for the centred data matrix, $\mathbf{\Sigma} = \frac{1}{n}\mathbf{B}\mathbf{B}^\mathsf{T}$:

$$\mathbf{\Sigma} = \frac{1}{5}\begin{bmatrix} 406.8333 & 176.3333 & 52.5 \\ 176.3333 & 103.3333 & 36.0 \\ 52.5000 & 36.0000 & 51.5 \end{bmatrix}.$$

This matrix has eigenvalues 99.31, 9.46 and 3.561 corresponding to eigenvectors

$$[\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3] = \begin{bmatrix} 0.8987 & 0.2829 & 0.3352 \\ 0.4158 & -0.3062 & -0.8564 \\ 0.1396 & -0.9090 & 0.3928 \end{bmatrix} = \mathbf{U}.$$

These eigenvectors are the principal components of $\mathbf{A}$ (and of $\mathbf{B}$).

The amount of the first principal component in $\mathbf{a}_1$ is

$$\mathbf{u}_1^\mathsf{T}\mathbf{a}_1 = [0.8987, 0.4158, 0.1396]\begin{bmatrix} -4 \\ 2 \\ 7 \end{bmatrix} = -1.756.$$

To project $\mathbf{A}$ into the coordinate system defined by the first two principal components, form the projection matrix,

$$\mathbf{P}_2 = \mathbf{U}_2\mathbf{U}_2^\mathsf{T} = \begin{bmatrix} 0.8987 & 0.2829 \\ 0.4158 & -0.3062 \\ 0.1396 & -0.9090 \end{bmatrix}\begin{bmatrix} 0.8987 & 0.4158 & 0.1396 \\ 0.2829 & -0.3062 & -0.9090 \end{bmatrix} = \begin{bmatrix} 0.8877 & 0.2870 & -0.1316 \\ 0.2870 & 0.2666 & 0.3364 \\ -0.1316 & 0.3368 & 0.8457 \end{bmatrix}$$

8.6 Examples

See associated slides for population structure in Europe (Novembre et al, Nature 2008, `http://www.nature.com/nature/journal/v456/n7218/full/nature07331.html`)and Eigenfaces.

The “eigenfaces” example in the slides was developed by Matthew Turk and Alex Pentland (Journal of Cognitive Neuroscience, 1991, v3 (1)). The following quote is from their abstract:

> We have developed a near-real-time computer system that can locate and track a subject’s head, and then recognize the person by comparing characteristics of the face to those of known individuals. ... The system functions by projecting face images onto a feature space that spans the significant variations among known face images. The significant features are known as ”eigenfaces,” because they are the eigenvectors (principal components) of the set of faces; they do not necessarily correspond to features such as eyes, ears, and noses. The projection operation characterizes an individual face by a weighted sum of the eigenface features, and so to recognize a particular face it is necessary only to compare these weights to those of known individuals. Some particular advantages of our approach are that it provides for the ability to learn and later recognize new faces in an unsupervised manner, and that it is easy to implement using a neural network architecture.

8.7 What is connection between PCA and SVD?

Given $\mathbf{A}$ such that the rows of $\mathbf{A}$ have zero mean, define $\mathbf{Y} = \frac{1}{\sqrt{n-1}}\mathbf{A}^T$ (which has columns with zero mean). Then $\mathbf{Y}^T\mathbf{Y} = \mathbf{\Sigma}$, the covariance of $\mathbf{A}$. We have seen that the principal components of $\mathbf{A}$ are the eigenvectors of $\mathbf{\Sigma}$.

Now, if we calculate the SVD of $\mathbf{Y}$ to get $\mathbf{Y} = \mathbf{UDV}^T$, the columns of $\mathbf{V}$ are the eigenvectors of $\mathbf{Y}^T\mathbf{Y} = \mathbf{\Sigma}$. Therefore, the columns of $\mathbf{V}$ are the principal components of $\mathbf{A}$.

8.8 Problems with SVD and PCA

As we have seen, SVD and PCA are powerful analysis tools and SVD is a very stable procedure. They do not, however, come free of cost.

The time complexity of SVD is $O(m^2n + n^3)$ to calculate all of $\mathbf{U}, \mathbf{V}$ and $\mathbf{D}$ (where, typically, $m \gg n$) while faster algorithms are available when some elements of the SVD are not required.

However, the matrices $\mathbf{U}$ and $\mathbf{V}$ are not at all *sparse*, where we say a matrix is sparse when it mainly consists of zeros. Spareness is a commonly assumed property in large systems as it reflects the observation that most effects are local and do not influence all parameters in the system — a large world with small neighbourhoods. Sparse matrices are typically computationally efficient to work with and store.

A second potential set-back is that SVD and PCA only work with data that can be (coherently) expressed as a two dimensional array (that is, a matrix). When data naturally has 3 or 4 dimensions arrays (*tensors*), as is common in many engineering applications, there is no perfect analogue to SVD or PCA or even eigenvectors.

Finally, when using PCA for data analysis, you should be aware of the strong assumptions being made. In particular, dependencies in the data are assumed to be linear, which may not be the case. PCA and SVD will always give an answer but it is up to the user to interpret whether or not it is a valid answer to any question they are interested in.

9 Least squares

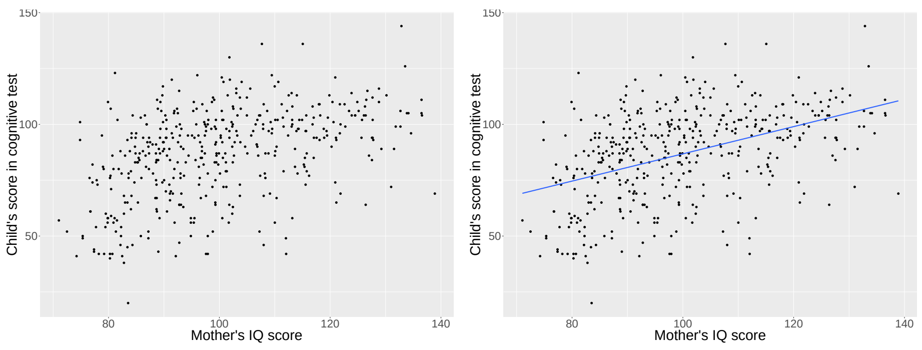


Figure 5: Left: Relationship between cognitive test scores for 3-4 year old children and mother's IQ score. Right: The same data with a least squares best fit line added. Discussed in Gelman and Hill, 2007, Cambridge University Press, data at http://www.stat.columbia.edu/ gelman/arm/examples/child.iq/kidiq.dta

You are probably familiar with the basic idea of least squares: we have a set of measurements and we want to fit a model to them. But no sufficiently simple model exactly fits all of the points at the same time. So how choose the model that is most satisfactory? The answer often given is that we chose the model that satisfies the *least squares* criterion: that is, the model for which the sum of the squares of differences between the predictions from the model and the actual observations is minimised.

For example, in Figure 5, we might want to fit a linear model to the relationship between a mother's IQ score and her young child's score in a cognitive test. This should be familiar to you as the linear regression problem in statistics.

This problem arises when we have an *overdetermined* linear system: recall that $\mathbf{Au} = \mathbf{b}$ is overdetermined when $\mathbf{A}$ is $m \times n$ matrix with $m > n$:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}.$$

In this case, $\mathbf{A}^{-1}$ does not exist and there is no $\mathbf{u}$ that solves this problem. (We ignore the highly unusual cases where a solution does exist.)

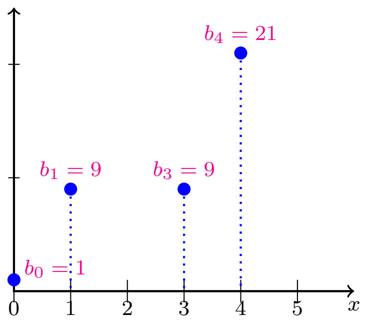
The goal, then, is to find the best solution $\mathbf{u}^*$ to the problem.

Example 1: Fitting $m = 4$ measurements by a small number $n = 2$ of parameters (e.g. linear regression in statistics)

Want to find the straight line $b_x = u_1 + u_2 x$ where we have observed the points b_x at x.

$$\left\{\begin{array}{lcl} u_1 + u_2 \cdot 0 & = & 1 \\ u_1 + u_2 \cdot 1 & = & 9 \\ u_1 + u_2 \cdot 2 & = & 9 \\ u_1 + u_2 \cdot 4 & = & 21 \end{array}\right. \Leftrightarrow$$

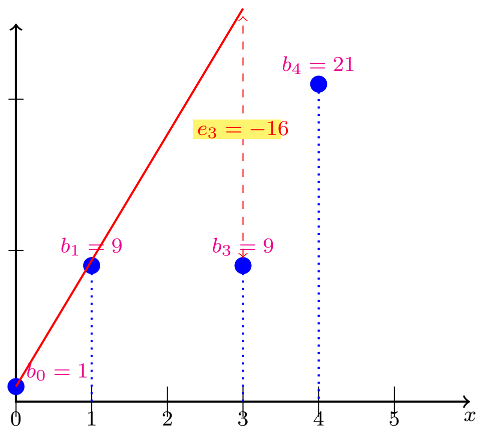
$$\left\{ \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 3 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 9 \\ 9 \\ 21 \end{bmatrix} \right.$$



The above set of equations clearly has no solution as vector **b** is not a linear combination of the two column vectors from **A**:

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 3 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \neq \begin{bmatrix} 1 \\ 9 \\ 9 \\ 21 \end{bmatrix}$$

For example, The line $b = 1 + 8x$ through the first two points is almost certainly not the best line:



But why is this not the best line: look at the *error* or *residual* , $\mathbf{e} = \mathbf{b} - \mathbf{A}\mathbf{u}$. For the two points the line does not pass through the error is $e_x = b_x - (1 + 8x)$ is large: $e_3 = 16$ and $e_4 = 12$. The *Total square error*, $E(\mathbf{u}) = 0 + 0 + 256 + 144 = 400$.

Notice that the *total square error* is given by.

$$E(\mathbf{u}) = \mathbf{e}^{\mathsf{T}}\mathbf{e} \equiv \| \mathbf{e} \|^2 = (\mathbf{b} - \mathbf{A}\mathbf{u})^{\mathsf{T}}(\mathbf{b} - \mathbf{A}\mathbf{u})$$

The Least Squares method to find the chooses a solution $\mathbf{u}^*$ that minimises $E(\mathbf{u})$.

How do we find $\mathbf{u}^*$? To find the minimum of $E(\mathbf{u})$, we can differentiate with respect to $\mathbf{u}$, set to 0 and attempt to solve for $\mathbf{u}$:

$$\begin{aligned} E(\mathbf{u}) &= (\mathbf{b} - \mathbf{A}\mathbf{u})^{\mathsf{T}}(\mathbf{b} - \mathbf{A}\mathbf{u}) \\ &= \mathbf{b}^{\mathsf{T}}\mathbf{b} - 2\mathbf{u}^{\mathsf{T}}\mathbf{A}^{\mathsf{T}}\mathbf{b} + \mathbf{u}^{\mathsf{T}}\mathbf{A}^{\mathsf{T}}\mathbf{A}\mathbf{u} \end{aligned}$$

Differentiating and setting to 0:

$$\begin{aligned} \tfrac{\partial E(\mathbf{u})}{\partial \mathbf{u}} &= 0 \\ \Longrightarrow \quad -2\mathbf{A}^{\mathsf{T}}\mathbf{b} + 2\mathbf{A}^{\mathsf{T}}\mathbf{A}\mathbf{u} &= \mathbf{0} \\ \Longrightarrow \quad \mathbf{A}^{\mathsf{T}}\mathbf{A}\mathbf{u} &= \mathbf{A}^{\mathsf{T}}\mathbf{b} \end{aligned}$$

This equation, $\mathbf{A}^{\mathsf{T}}\mathbf{A}\mathbf{u} = \mathbf{A}^{\mathsf{T}}\mathbf{b}$ is called the *normal equation.*

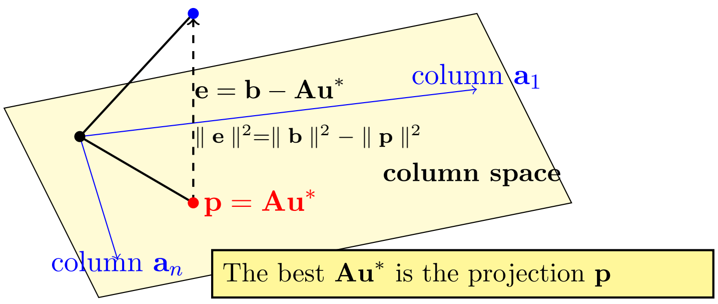
The least squares estimate, $\mathbf{u}^*$, is the solution to the normal equation.

Notice that $\mathbf{A}^T\mathbf{A}$ is square and symmetric. In some cases it may be possible to directly find the inverse (in particular, when $\mathbf{A}$ has independent columns, then $\mathbf{A}^T\mathbf{A}$ is positive definite and $\mathbf{A}^T\mathbf{A}$ is invertible in which case $\mathbf{u}^* = (\mathbf{A}^{\mathsf{T}}\mathbf{A})^{-1}\mathbf{A}^{\mathsf{T}}\mathbf{b}$). In other cases, this approach may be highly unstable, so stable numerical techniques need to be employed.

9.1 Understanding the Least Squares solution

This subsection is not examined. The main point of this section is to add some geometric and algebraic understanding to our discussion. You are not expected to understand all the detail in this section, but do familiarise yourself with the concept and definition of the projection matrix $\mathbf{P}$ defined below.

The equation $\mathbf{A}\mathbf{u} = \mathbf{b}$ can be seen as attempting to represent $\mathbf{b}$ as a linear combination of the n columns of $\mathbf{A}$. This is impossible, since the n columns of $\mathbf{A}$ describe, at most, an n-dimensional plane inside the much larger m dimensional space (recall that $n < m$). Thus b is unlikely to fall on that plane. The plane is called the *column space* of $\mathbf{A}$.



$\mathbf{e} = \mathbf{b} - \mathbf{A}\mathbf{u}^*$
column $\mathbf{a}_1$
$\| \mathbf{e} \|^2 = \| \mathbf{b} \|^2 - \| \mathbf{p} \|^2$
column space
$\mathbf{p} = \mathbf{A}\mathbf{u}^*$
column $\mathbf{a}_n$
The best $\mathbf{A}\mathbf{u}^*$ is the projection $\mathbf{p}$

The best solution, $\mathbf{A}\mathbf{u}^*$, is the nearest point to $\mathbf{b}$ on that plane. Call this point $\mathbf{p} = \mathbf{A}\mathbf{u}^*$. Now, from a geometric argument, you can see that the error vector $\mathbf{e}$ is orthogonal (perpendicular) to this plane. Thus $\mathbf{A}^T e = 0$.

Notice that $0 = \mathbf{A}^T\mathbf{e} = \mathbf{A}^T(\mathbf{b} - \mathbf{A}\mathbf{u}^*) = \mathbf{A}^T\mathbf{b} - \mathbf{A}^T\mathbf{A}\mathbf{u}^* \implies \mathbf{A}^T\mathbf{b} = \mathbf{A}^T\mathbf{A}\mathbf{u}^*$. This is a geometric derivation of the normal equation that we earlier saw derived from calculus.

The point $\mathbf{p}$ $(= \mathbf{A}\mathbf{u}^*)$ is the projection of $\mathbf{b}$ onto the column space of $\mathbf{A}$:

$$\mathbf{p} = \mathbf{A}\mathbf{u}^* = \underbrace{\left[\mathbf{A}\left(\mathbf{A}^\mathsf{T}\mathbf{A}\right)^{-1}\mathbf{A}^\mathsf{T}\right]}_{\text{projection matrix } \mathbf{P}}\mathbf{b} = \mathbf{P}\mathbf{b},$$

where we define the *Projection matrix*, $\mathbf{P} = \mathbf{A}\left(\mathbf{A}^\mathsf{T}\mathbf{A}\right)^{-1}\mathbf{A}^\mathsf{T}$. $\mathbf{P}$ is symmetric and of size $m \times m$ but the rank of $\mathbf{P}$ is only n (as all the factors of $\mathbf{P}$ in the definition above have rank n).

9.2 Computing the Least Squares solution, $\mathbf{u}^*$

We consider three methods for computing the least squares solution to a linear system. They are Gaussian elimination, QR Decomposition (aka Orthogonalisation) and computation of the pseudo-inverse via SVD.

9.3 Computing $\mathbf{u}^*$ via Gaussian elimination

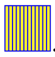
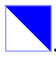
Given the normal equation $\mathbf{A}^T\mathbf{A}\mathbf{u} = \mathbf{A}^T\mathbf{b}$, we may be tempted to find the solution by Gaussian elimination, where we reduce the the matrix $\mathbf{A}^T\mathbf{A}$ to upper triangular form using elementary row operations.

This solution can work but is highly unstable. To see why it is unstable, consider the condition number of the matrix $\mathbf{A}^T\mathbf{A}$. It can be shown that the condition number of $\mathbf{A}^T\mathbf{A}$ is the square of the condition number of $\mathbf{A}$ (if we take σ_{min} to be the smallest non-zero singular value in the definition of condition number). So even if $\mathbf{A}$ has only moderately widely spread singular values, $\mathbf{A}^T\mathbf{A}$ can have a very large condition number and solution by row reduction can be very unstable.

9.4 Computing $\mathbf{u}^*$ via orthogonalisation (QR decomposition)

QR-decompostion presents a much more stable solution to the normal equation $\mathbf{A}^\mathsf{T}\mathbf{A}\mathbf{u} = \mathbf{A}^\mathsf{T}\mathbf{b}$.

The Orthogonalisation of matrix $\mathbf{A}$ is given by $\mathbf{A} = \mathbf{Q}\mathbf{R}$ where

- $\mathbf{Q}$ is an $m \times n$ matrix with n orthonormal columns: . Construction of $\mathbf{Q}$ is discussed below.
- $\mathbf{R}$ is an $n \times n$ upper triangular matrix: . $\mathbf{R}$ is given by $\mathbf{R} = \mathbf{Q}^\mathsf{T}\mathbf{A}$.

This factorisation reduces the normal equation to a much simpler equation:

$$\begin{aligned}
\mathbf{A}^\mathsf{T}\mathbf{A}\mathbf{u} &= \mathbf{A}^\mathsf{T}\mathbf{b} \\
\Longrightarrow (\mathbf{Q}\mathbf{R})^\mathsf{T}\mathbf{Q}\mathbf{R}\mathbf{u}^* &= (\mathbf{Q}\mathbf{R})^\mathsf{T}\mathbf{b} \\
\Longrightarrow \mathbf{R}^\mathsf{T}\mathbf{Q}^\mathsf{T}\mathbf{Q}\mathbf{R}\mathbf{u}^* &= \mathbf{R}^\mathsf{T}\mathbf{Q}^\mathsf{T}\mathbf{b} \\
\Longrightarrow \mathbf{R}^\mathsf{T}\mathbf{R}\mathbf{u}^* &= \mathbf{R}^\mathsf{T}\mathbf{Q}^\mathsf{T}\mathbf{b} \text{ since } \mathbf{Q}^\mathsf{T}\mathbf{Q} = \mathbf{I} \\
\Longrightarrow \mathbf{R}\mathbf{u}^* &= \mathbf{Q}^\mathsf{T}\mathbf{b} \text{ multiplying both sides by } (\mathbf{R}^\mathsf{T})^{-1}.
\end{aligned}$$

This is easy to solve via back-substitution, since $\mathbf{R}$ is upper triangular.

9.4.1 Constructing the orthogonal matrix Q by Gram-Schmidt

The orthonormal columns of $\mathbf{Q}$, call them $\mathbf{q}_1, \dots, \mathbf{q}_n$, are obtained iteratively from the columns $\mathbf{a}_1, \dots, \mathbf{a}_n$ of $\mathbf{A}$. The basic idea is that we set $\mathbf{q}_1$ to be $\mathbf{a}_1$. q_2 is then set to be a_2 and any part of it in the direction of $q_1 (= a_1)$ is subtracted out, so ensure that it is orthogonal to q_1. Similarly, q_3 is set to be a_3 with any parts in the direction of q_1 or q_2 are subtracted. All these vectors are normalised to have magnitude 1 at each step.

This is called the *Gram-Schmidt* process and is more formally defined as follows:

$$\begin{array}{llll}
\text{Set } \mathbf{v}_1 = \mathbf{a}_1. & & \text{Then} & \mathbf{q}_1 = \frac{\mathbf{v}_1}{|\mathbf{v}_1|}. \\
\text{Set } \mathbf{v}_2 = \mathbf{a}_2 - \left(\mathbf{a}_2^\mathsf{T}\mathbf{q}_1\right)\mathbf{q}_1. & & \text{Then} & \mathbf{q}_2 = \frac{\mathbf{v}_2}{|\mathbf{v}_2|}. \\
\vdots \quad \vdots & & & \vdots \quad \vdots \\
\text{Set } \mathbf{v}_j = \mathbf{a}_j - \sum_{i=1}^{j-1}\left(\mathbf{a}_j^\mathsf{T}\mathbf{q}_i\right)\mathbf{q}_i. & & \text{Then} & \mathbf{q}_j = \frac{\mathbf{v}_j}{|\mathbf{v}_j|}.
\end{array}$$

Note that $|\mathbf{v}| = \mathbf{v}^\mathsf{T}\mathbf{v}$ is the norm of $\mathbf{v}$.

Having found $\mathbf{Q}$, find $\mathbf{R}$ by setting $\mathbf{R} = \mathbf{Q}^\top\mathbf{A}$. $\mathbf{R}$ is indeed upper triangular since the ith column of $\mathbf{Q}$ is, by construction, orthogonal to first $i-1$ columns of $\mathbf{A}$.

Producing $\mathbf{Q}$ and $\mathbf{R}$ takes twice as long as the mn^2 steps to form $\mathbf{A}^\mathsf{T}\mathbf{A}$, but that extra cost gives a more reliable solution.

There is another method of orthogonalisation that we don't cover here which has better numerical stability using so-called Householder reflectors.

Example: Use the Gram-Schmidt process to orthogonalise the matrix

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

Solution: Let $\mathbf{v}_1 = \mathbf{a}_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$ and then normalise to get $\mathbf{q}_1 = \frac{\mathbf{v}_1}{|\mathbf{v}_1|} = \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \\ 0.5 \end{bmatrix}$. Now set $\mathbf{v}_2 = \mathbf{a}_2 - (\mathbf{a}_2^\top \mathbf{q}_1)\mathbf{q}_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} - \left(\begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \\ 0.5 \end{bmatrix} \right) \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \\ 0.5 \end{bmatrix} = \begin{bmatrix} -0.5 \\ -0.5 \\ 0.5 \\ 0.5 \end{bmatrix}$.

Since $\mathbf{v}_2 = 1$, $\mathbf{q}_2 = \frac{\mathbf{v}_2}{|\mathbf{v}_2|} = \mathbf{v}_2$.

Finally, to get $\mathbf{q}_3$, set

$$\begin{aligned}
\mathbf{v}_3 &= \mathbf{a}_3 - (\mathbf{a}_3^\top \mathbf{q}_1)\mathbf{q}_1 - (\mathbf{a}_3^\top \mathbf{q}_2)\mathbf{q}_2 \\
&= \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} - \left(\begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \\ 0.5 \end{bmatrix} \right) \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \\ 0.5 \end{bmatrix} - \left(\begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} -0.5 \\ -0.5 \\ 0.5 \\ 0.5 \end{bmatrix} \right) \begin{bmatrix} -0.5 \\ -0.5 \\ 0.5 \\ 0.5 \end{bmatrix} \\
&= \begin{bmatrix} 0.5 \\ -0.5 \\ 0.5 \\ -0.5 \end{bmatrix}
\end{aligned}$$

which is also normalised, so $\mathbf{q}_3 = \mathbf{v}_3$ and

$$\mathbf{Q} = \frac{1}{2} \begin{bmatrix} 1 & -1 & 1 \\ 1 & -1 & -1 \\ 1 & 1 & 1 \\ 1 & 1 & -1 \end{bmatrix}.$$

We find $\mathbf{R}$ as follows:

$$\mathbf{R} = \mathbf{Q}^\mathsf{T}\mathbf{A} = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}.$$

□

9.5 Computing $\mathbf{u}^*$ via SVD: the Pseudoinverse

The most stable computation to find the solution to the normal equation is given by singular value decomposition (SVD).
Recall that SVD decomposes the $m \times n$ matrix $\mathbf{A}$ as $\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^\mathsf{T}$ where:

- $\mathbf{U}$ is a column-orthonormal $n \times m$ matrix, so $\mathbf{U}^\mathsf{T}\mathbf{U} = \mathbf{I}_n$,
- $\mathbf{V}$ is an orthonormal $n \times n$ matrix so $\mathbf{V}^\mathsf{T}\mathbf{V} = \mathbf{I}_n$ (indeed, $\mathbf{V}^\mathsf{T} = \mathbf{V}^{-1}$), and
- $\mathbf{D} = \mathrm{diag}\{\sigma_1, \ldots, \sigma_n\}$ is a diagonal $n \times n$ matrix of singular values. Since $\mathbf{D}$ is diagonal, $\mathbf{D}^\mathsf{T} = \mathbf{D}$.

Now consider the product $\mathbf{A}^\mathsf{T}\mathbf{A}$ that arises in the normal equation. Substituting $\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^\mathsf{T}$ in this product gives:

$$\mathbf{A}^\mathsf{T}\mathbf{A} = (\mathbf{U}\mathbf{D}\mathbf{V}^\mathsf{T})^\mathsf{T}\mathbf{U}\mathbf{D}\mathbf{V}^\mathsf{T} = \mathbf{V}\mathbf{D}^\mathsf{T}\mathbf{U}^\mathsf{T}\mathbf{U}\mathbf{D}\mathbf{V}^\mathsf{T} = \mathbf{V}\mathbf{D}^\mathsf{T}\mathbf{D}\mathbf{V}^\mathsf{T} = \mathbf{V}\mathbf{D}^2\mathbf{V}^\mathsf{T}.$$

We can thus express the normal equation in a much simplified form:

$$\begin{aligned}
\mathbf{A}^\mathsf{T}\mathbf{A}\mathbf{u} &= \mathbf{A}^\mathsf{T}\mathbf{b} \\
\Longrightarrow \mathbf{V}\mathbf{D}^2\mathbf{V}^\mathsf{T}\mathbf{u}^* &= \mathbf{V}\mathbf{D}\mathbf{U}^\mathsf{T}\mathbf{b} \\
\Longrightarrow \mathbf{D}^2\mathbf{V}^\mathsf{T}\mathbf{u}^* &= \mathbf{D}\mathbf{U}^\mathsf{T}\mathbf{b} \\
\Longrightarrow \mathbf{V}^\mathsf{T}\mathbf{u}^* &= \underbrace{\left(\mathbf{D}^2\right)^{-1}\mathbf{D}}_{\mathbf{D}^+}\mathbf{U}^\mathsf{T}\mathbf{b} \\
\Longrightarrow \mathbf{u}^* &= \mathbf{V}\mathbf{D}^+\mathbf{U}^\mathsf{T}\mathbf{b}.
\end{aligned}$$

The matrix $\mathbf{D}^+$ is called the "*pseudoinverse*" of $\mathbf{D}$ and is defined as follows: $\mathbf{D}^+ = \mathrm{diag}\left\{\sigma_1^+, \ldots, \sigma_n^+\right\}$ where

$$\sigma_i^+ = \begin{cases} \sigma_i^{-1} = \frac{1}{\sigma_i} & \text{if} \quad \sigma_i > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Thus if $\mathrm{rank}(\mathbf{A}) = n$, then all the singular values of $\mathbf{A}$ are non-zero, in which case $\mathbf{D}^+ = \mathbf{D}^{-1} = \mathrm{diag}\left\{\frac{1}{\sigma_1}, \ldots, \frac{1}{\sigma_n}\right\}$. In this case, In the former case, $\mathbf{D}\mathbf{D}^+ = \mathbf{D}^+\mathbf{D} = \mathbf{I}_n$.
However, if $\mathrm{rank}(\mathbf{A}) = r < n$, there are only $r < n$ non-zero singular values and $\mathbf{D}^+ = \mathrm{diag}\{\frac{1}{\sigma_1}, \ldots, \frac{1}{\sigma_k}, \underbrace{0, \ldots, 0}_{n-r \text{ zeros}}\}$. In this case, $\mathbf{D}\mathbf{D}^+ = \mathbf{D}^+\mathbf{D} = \mathrm{diag}\{1, \ldots, 1, \underbrace{0, \ldots, 0}_{n-r \text{ zeros}}\}$ — which is very close to, but not quite, the identity matrix.
We call the product matrix $\mathbf{V}\mathbf{D}^+\mathbf{U}^\mathsf{T}$ the *pseudoinverse* of $\mathbf{A}$, written $\mathbf{A}^+$. That is ,

$$\mathbf{A}^+ = \mathbf{V}\mathbf{D}^+\mathbf{U}^\mathsf{T}.$$

If $\mathrm{rank}(\mathbf{A}) = n$, then $\mathbf{A}^+\mathbf{A} = \mathbf{V}\mathbf{D}^+\mathbf{U}^\mathsf{T}\mathbf{U}\mathbf{D}\mathbf{V}^\mathsf{T} = \mathbf{V}\mathbf{D}^+\mathbf{D}\mathbf{V}^\mathsf{T} = \mathbf{V}\mathbf{V}^\mathsf{T} = \mathbf{I}_n$, and $\mathbf{A}^+ = \mathbf{A}^{-1}$.

Recall that the matrix $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ is ill-conditioned when the smallest singular value, σ_n, is very small. This leads to instability in computing the solution to the normal equation. The pseudo-inverse method provides a way of removing this instability to get an approximate but stable solution by simply removing the smallest singular value or values.

9.5.1 Properties of the pseudo inverse $\mathbf{A}^+$

The pseudo-inverse of $\mathbf{A}$ always exists (all we need to do is calculate the SVD and form the product described above).
The SVD of $\mathbf{A}$ gives $\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^{\mathrm{T}}$ from which we get $\mathbf{A}\mathbf{V} = \mathbf{U}\mathbf{D}$ or, considering the individual columns, $\mathbf{A}\mathbf{v}_i = \sigma_i \mathbf{u}_i$.

- If $\mathbf{A}$ is a square matrix such that $\mathbf{A}^{-1}$ exists, then the singular values for $\mathbf{A}^{-1}$ are $\sigma^{-1} = \frac{1}{\sigma}$ and $\mathbf{A}^{-1}\mathbf{u}_i = \frac{1}{\sigma_i}\mathbf{v}_i$
- If $\mathbf{A}^{-1}$ does not exist, then the pseudoinverse matrix $\mathbf{A}^+$ does exist such that:
$$\mathbf{A}^+\mathbf{u}_i = \begin{cases} \frac{1}{\sigma_i}\mathbf{v}_i & \text{if} \quad i \le r = \operatorname{rank}(\mathbf{A}) \text{ i.e. if } \sigma_i > 0 \\ 0 & \text{for} \quad i > r. \end{cases}$$
- Pseudoinverse matrix $\mathbf{A}^+$ has the same rank r as $\mathbf{A}$
- The matrices $\mathbf{A}\mathbf{A}^+$ and $\mathbf{A}^+\mathbf{A}$ are also as near as possible to the $m \times m$ and $n \times n$ identity matrices, respectively
- $\mathbf{A}\mathbf{A}^+$ – the $m \times m$ projection matrix onto the column space of $\mathbf{A}$
- $\mathbf{A}^+\mathbf{A}$ – the $n \times n$ projection matrix onto the row space of $\mathbf{A}$

Example: Find the pseudo-inverse of $\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 1 & 0 \end{bmatrix}$.

Solution: The singular value decomposition of $\mathbf{A}$ is

$$\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^{\top} = \underbrace{\begin{bmatrix} \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{2}} \\ \frac{2}{\sqrt{6}} & 0 \\ \frac{1}{\sqrt{6}} & -\frac{1}{\sqrt{2}} \end{bmatrix}}_{\mathbf{U}} \underbrace{\begin{bmatrix} \sqrt{3} & 0 \\ 0 & 1 \end{bmatrix}}_{\mathbf{D}} \underbrace{\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}}_{\mathbf{V}^{\mathrm{T}}}.$$

So the pseudo-inverse of $\mathbf{A}$ is

$$\mathbf{A}^+ = \mathbf{V}\mathbf{D}^+\mathbf{U}^{\top} = \underbrace{\begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}}_{\mathbf{V}} \underbrace{\begin{bmatrix} \frac{1}{\sqrt{3}} & 0 \\ 0 & 1 \end{bmatrix}}_{\mathbf{D}^+} \underbrace{\begin{bmatrix} \frac{1}{\sqrt{6}} & \frac{2}{\sqrt{6}} & \frac{1}{\sqrt{6}} \\ \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \end{bmatrix}}_{\mathbf{U}^{\mathrm{T}}}$$
$$= \begin{bmatrix} -\frac{1}{3} & \frac{1}{3} & \frac{2}{3} \\ \frac{2}{3} & \frac{1}{3} & -\frac{1}{3} \end{bmatrix}.$$

Now let's check the products $\mathbf{AA}^+$ and $\mathbf{A}^+\mathbf{A}$:

$$\mathbf{AA}^+ = \begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} -\frac{1}{3} & \frac{1}{3} & \frac{2}{3} \\ \frac{2}{3} & \frac{1}{3} & -\frac{1}{3} \end{bmatrix} = \begin{bmatrix} \frac{2}{3} & \frac{1}{3} & -\frac{1}{3} \\ \frac{1}{3} & \frac{2}{3} & \frac{1}{3} \\ -\frac{1}{3} & \frac{1}{3} & \frac{2}{3} \end{bmatrix}$$

while

$$\mathbf{A}^+\mathbf{A} = \begin{bmatrix} -\frac{1}{3} & \frac{1}{3} & \frac{2}{3} \\ \frac{2}{3} & \frac{1}{3} & -\frac{1}{3} \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

$\square$

We end our discussion on computational methods in linear algebra here. We have only touched on a small number of the wide variety of techniques used in this area, but we have tried to direct our attention to some of the more common and useful techniques. This is a rich and extremely useful area of study and we encourage interested students to look into fields where these tools and ideas are applied and explored further including computer vision, engineering, physics, applied mathematics and statistics.

10 Introduction to stochastic processes and probability

Models and methods that have been considered so far in the course have been deterministic — a single input produces the same answer every time while solutions to problems are aimed at finding one correct answer or a close approximations to it. These deterministic models, methods and approximations can be very accurate, particularly in engineering and physical applications.

Many systems, however, are inherently random. Apparently identical inputs may produce radically different outputs and no two realisations of the system are exactly the same. Whether that randomness is the result of our imprecise measurements (were the inputs exactly the same?) or there is a fundamental randomness built into the system (quantum uncertainty?), we should attempt to model and quantify this uncertainty. We regularly refer to non-deterministic systems as *stochastic* rather than random to avoid the common usage of random (where it is often used to mean uniformly random where every outcome is equally likely).

The framework we use to make these models is probability theory and, ideally, we use statistical inference to find the relationship between our models and the system we are studying. Simulation is one of the tools we use to understand how our models behave and is often used where exact statistical inference is prohibitively difficult. Both statistical inference and simulation rely heavily on computational power and algorithms. Model building is typically more of an art than a science and is done by hand (or mind).

In this section, we look at some of the basic terminology of probability theory, introduce the fundamental ideas behind statistical inference and see how we can simulate stochastic processes *in silico.*

11 Primer on Probability

The basic challenge of probability theory and applied probability is to understand and describe the laws according to which events occur.

A event can be pretty much anything. Commonly used examples in probability are rolling dice, picking balls out of an urn or tossing a coin — these are commonly used because they are simple, easy to understand and aid our intuition. But all sorts of events can be thought of as random: the amount of rain falling in an area in a given period, the number of mutations that occur when a cell splits, the age of the person currently reading this sentence. Indeed, if we consider randomness to a a property of our state of knowledge of an event, any event can be considered random.

Formally, we define a *probability* to be a number between 0 and 1 assigned to a set of outcomes of a random process called an *event.* This number is typically interpreted as the chance of the event occurring or as the degree of plausibility we place on the event occurring.

The set of all outcomes that the random process can take is known as the *state space* and is often denoted Ω.

An *event*, A, is a subset of the state space: $A \subseteq \Omega$. When Ω is finite or countable, probability can be viewed as a function from the set of all subsets of Ω, written $\mathcal{A}$, to the interval $[0,1]$, that is $P : \mathcal{A} \to [0,1]$. $\mathcal{A}$, the set of all subsets of Ω, is called the *power set* of Ω. That is, for some event or collection of events, we view the function P as giving the probability of that event occurring.

This interpretation is not mathematically correct when Ω is uncountable (for example, when our random process can take any value in continuous interval) but it will be sufficient to guide our intuition here.

Example: Tossing a coin 2 times and recording the result of each toss. The random process is tossing the coin twice. The state space is the set of all possible outcomes: $\Omega = \{HH, HT, TH, TT\}$.

An example of an event is that we throw a tails first: in this case $A = \{TH, TT\}$.

There are $2^4 = 16$ possible events that we could consider here as that is the size of the power-set (the set of all possible subsets) of Ω. For completeness, we write down all possible events: $\mathcal{A} = \{\emptyset, \{HH\}, \{HT\}, \{HT\}, \{TT\}, \{HH, HT\}, \{HH, TH\}, \{HH, TT\}, \{HT, TH\}, \{HT, TT\}, \{TH, TT\}, \{HH, HT, TH\}, \{HH, HT, TT\}, \{HH, TH, TT\}, \{HT, TH, TT\}, \Omega\}$. $\square$

Note that sometimes we distinguish between simple and compound events. In the above example, the simple events are $\{HH\}, \{HT\}, \{HT\}, \{TT\}$ while compound events are combinations of simple events.

Example: Length of time waiting for bus, measured from arrival at bus stop until bus arrives. Supposing the buses come every 15 mins. Then $\Omega = [0, 15]$ (that is, any time in the interval between 0 and 15 minutes). An example of an event is $A = [0, 1]$ being the event that the wait for the bus is at most 1 minute. $\square$

Let A and B be events. Then the event C that A and B occur is given by $C = A \cap B$ while the event F that A or B occurs is given by $D = A \cup B$. The event A does not occur is given by $A^c = \bar{A} = \Omega - A = \Omega \backslash A = \{\omega \in \Omega : \omega \notin A\}$.

Example: Suppose we roll a fair die and record the value. Then $\Omega = \{1, 2, 3, 4, 5, 6\}$. Let A be the event that the roll is even, B be the event that we roll a 3 or a 6. Then $A = \{2, 4, 6\}$ and $B = \{3, 6\}$. The event that A and B occur is $A \cap B = \{6\}$ while the event that A or B happens is $A \cup B = \{2, 3, 4, 6\}$. The event that A does not occur is $A^c = \{1, 3, 5\}$ (A does not occur when the roll is odd). $\square$

11.1 Axioms of probability

Any probability function must satisfy the 3 axioms (rules) of probability. The axioms are:

1. $P(\Omega) = 1$. That is, the total probability is 1.

2. $0 \leq P(A) \leq 1$ for any $A \subseteq \Omega$. The probability of any event is non-negative and

less than or equal to 1.

3. If $A_1, A_2, \ldots$ are mutually disjoint events (i.e., $A_i \cap A_j = \emptyset$ if $i \neq j$) then

$$P\left(\bigcup_i A_i\right) = \sum_i P(A_i).$$

From the above axioms, all the useful rules of probability can be derived. For example, $P(A^c) = 1 - P(A)$ since $1 = P(\Omega)$ (axiom 1) $= P(A \cup A^c)$ (definition of A^c) $= P(A) + P(A^c)$ (axiom 3 as A and A^c are disjoint).

11.2 Conditional probability and independence

For events A and B, if we know that B occurred, what can we say about the probability of A given that knowledge? This is captured by the concept of the *conditional probability of A given B* is written $P(A|B)$ and defined by

$$P(A|B) = \frac{P(A \cap B)}{P(B)}.$$

This is only defined where $P(B) > 0$.

Example: Suppose we roll a fair die and record the value. Let A be the event that 2 is rolled and B be the event that the roll is an even number. What is the probability that the roll is a two given that we know it is even? This is just $P(A|B)$. We calculate it as follows. $P(B) = 1/2$ and $P(A \cap B) = P(A) = 1/6$ since $A \cap B = A$. Thus

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{1/6}{1/2} = \frac{1}{3}.$$

□

We say that events A and B are *independent* when $P(A \cap B) = P(A)P(B)$. From the definition of conditional probability, it is clear that if A and B are independent, then $P(A|B) = P(A)$ and $P(B|A) = P(A)$.

Note that we usually write $P(A, B)$ instead of $P(A \cap B)$. More generally, we write $P(A_1, \ldots, A_k)$ for $P(\bigcap_{i=1}^k A_i)$.

Rearranging the definition of conditional probability, we see that $P(A, B) = P(A|B)P(B) = P(B|A)P(B)$. Repeated applications of this result gives

$$P(A_1, \ldots, A_k) = P(A_1|A_2, \ldots, A_k)P(A_2|A_3, \ldots, A_k) \ldots P(A_{k-1}|A_k)P(A_k).$$

11.3 Bayes' Theorem

From the definition of conditional probability, we can prove the following result, known as Bayes' theorem.

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)}.$$

This simple result is important because it tells us how the forward probability $P(A|B)$ is related to the backward probability $P(B|A)$. We'll see that this relationship is crucial to statistical inference.

11.4 Random variables

A *random variable* (r.v.) X is a variable whose value results from the measurement of a random process. That is, a random variable is a measurement of some random event. We use capital letters to denote random variables while lower-case letters to denote particular observations or *realisations* of the random variable. So $X = x$ is the event that the random variable X takes the particular value x.

A **discrete** random variable takes a finite or countably infinite number of values, while a **continuous** random variable can take an uncountable number of possible values.

Random variables are most commonly real valued (that is, their value is a real number) but they can take any value. For example, we could consider random sequences, random graphs or random trees. For now, lets stick with real valued random variables. Formally, a real-valued random variable is a map from events to the real numbers: $X : \Omega \to \mathbb{R}$.

For discrete random variables, the **probability distribution function** (pdf) or **probability mass function** (pmf) is a function (rule, table) that assigns probabilities to each possible value of X.

$P(X = x) = p(x)$ is the probability that $X = x$. Sometimes write $P(X = x) = p_x$ or $P(X = x) = f(x)$.

As usual, we have $0 \leq P(X = x) \leq 1$ and $\sum_x P(X = x) = 1$.

For continuous random variables, the probability that a random variable takes any one exact value is zero, that is $P(X = x) = 0$, so we consider instead the **probability density function**, $p_X(x)$ (also written $f_X(x)$ or $f(x)$) from which we can calculate the probability that X lies in the interval $[a, b]$:

$$Pr(a \leq X \leq b) = \int_a^b p_X(x)dx.$$

$p_X(x)$ is real-valued, non-negative and normalised, i.e.,

$$\int_{-\infty}^{\infty} p_x(x)dx = 1.$$

Note that the integral $\int_a^b p_X(x)dx$ gives the area under the curve $p_X(x)$ between $x = a$ and $x = b$.

The **cumulative distribution function** (cdf), or simply the *distribution function*, is defined, for both discrete and continuous random variables, by $F(x) = P(X \leq x)$. This is a function that is monotonically increasing from 0 to 1. For continuous random variables, the cdf is continuous, while for discrete random variables, it is a step function with dis-continuities.

These ideas immediately extend to multiple random variables, so that the **joint probability density function** of n random valuables $X_1, \ldots, X_n$ takes n arguments, $p_{X_1,\ldots,X_n}(x_1, \ldots, x_n)$ that is real-valued, non-negative and normalised. The probability that the point $(X_1, \ldots, X_n)$ lies in some region is just the multiple integral over that region.

Given a joint probability density function, we obtain the probability density for a subset of the variables by integrating over the ones not in the subset. For example, given $p_{XY}(x, y)$, we have

$$p_X(x) = \int_{-\infty}^{\infty} p_{XY}(x, y)dy.$$

This process is known as **marginalization**. The process is the same for a discrete variable, if we replace the integral with a sum:

$$P(x) = \sum_{y \in \mathcal{Y}} P(x, y)$$

Two random variables X and Y are **independent** when

$$p_{XY}(x, y) = p_X(x)p_Y(y).$$

Equivalently, X and Y are independent when

$$p_{Y|X}(y|x) = p_Y(y).$$

The **expected value** of a random variable X is called the mean and is given by

$$E[X] = \int_{-\infty}^{\infty} x p_X(x)dx.$$

For discrete random variables, this is written

$$E[X] = \sum_{x \in \mathcal{X}} x p_x.$$

The symbol μ is often used for the mean.

The **variance** of a random variable is $\mathrm{Var}(X) = E[(X - E[X])^2] = E[X^2] - (E[X])^2$. The variance is a measure of the spread of a random variable about its mean.

The **expected value of a function f of X** is

$$E[f(X)] = \int_{-\infty}^{\infty} f(x)p_X(x)dx.$$

11.5 Commonly used distributions

In this course, we'll primarily be discussing bioinformatics where some commonly used discrete probability distribution functions are: Bernoulli, geometric, binomial, uniform and Poisson. Commonly used continuous distributions are uniform, normal (Gaussian), exponential, and gamma. Those are briefly described below. For more thorough descriptions, refer to any decent statistics text or, more simply, the relevant Wikipedia entries.

11.5.1 Bernoulli distribution

A random variable X with a *Bernoulli distribution* takes values 0 and 1. It takes the value 1 on a 'success' which occurs with probability p where $0 \leq p \leq 1$. It takes value 0 on a failure with probability $q = 1 - p$. Thus it has the single parameter p. If X is Bernoulli, $E[X] = q \cdot 0 + p \cdot 1 = p$ and $\text{Var}(X) = E[X^2] - E[X]^2 = q \cdot 0^2 + p \cdot 1^2 - p^2 = pq$.

11.5.2 Geometric distribution

X has a *geometric distribution* when it is the number of Bernoulli trials that fail before the first success. It therefore takes values in $\{0, 1, 2, 3, \ldots\}$. If the Bernoulli trials have probability p of success, the pdf for X is $P(X = x) = (1 - p)^x p$. If X is geometric,

$$E[X] = \frac{q}{p} \text{ and } \text{Var}(X) = \frac{q}{p^2}.$$

Note that the Geometric distribution can be defined instead as the total number of trials required to get a single success. This version of the geometric can only take values in $\{1, 2, 3, \ldots\}$. The pdf, mean and variance all need to be adjusted accordingly.

11.5.3 Binomial distribution

X has a *binomial distribution* when it represents the number of successes in n Bernoulli trials. There are thus two parameters required to describe a binomial random variable: n, the number of Bernoulli trials undertaken, and p, the probability of success in the Bernoulli trials. The pdf for X is

$$f(x) = P(X = x) = \binom{n}{x} p^x (1 - p)^{n-x} \text{ for } x = 0, 1, 2, \ldots, n.$$

where $\binom{n}{x} = \frac{n!}{x!(n-x)!}$. For a binomial variable X,

$$E[X] = np \text{ and } Var[X] = np(1 - p) = npq.$$

We write $X \sim Bin(n, p)$ when X has a binomial distribution with parameters n and p.

11.5.4 Poisson distribution

The *Poisson distribution* is used to model the number of rare events that occur in a period of time. The events are considered to occur independently of each other. The distribution has a single parameter, λ, and probability density function

$$f(x) = \exp(-\lambda) \frac{\lambda^x}{x!} \text{ for } x = 0, 1, 2, 3, \ldots.$$

If X is Poisson,

$$E[X] = \lambda \text{ and } Var[X] = \lambda.$$

We write $X \sim Poiss(\lambda)$ when X has a Poisson distribution with parameter λ.

11.5.5 Uniform distribution (discrete or continuous)

Under the *uniform distribution*, all possible values are equally likely. So if X is discrete and takes n possible values, $P(X = x_i) = 1/n$ for all x_i.
If X is continuous and uniform over the interval $[a, b]$, the density function is $f(x) = \frac{1}{b-a}$. In this case, write $X \sim U([a, b])$.

11.5.6 Normal distribution

The *Normal*, or Gaussian, distribution, with mean μ and variance σ^2, $(\mu \in \mathbb{R}; \sigma > 0)$ has density function

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\frac{1}{2\sigma^2}(x-\mu)^2\right\}$$

We write $X \sim N(\mu, \sigma^2)$.
The normal distribution is a widely used distribution in statistical modelling for a number of reasons. A primary reason is that it arises as a consequence of the central limit theorem which says that (under a few weak assumptions) the sum of a set of identical random variables is well approximated by a normal distribution. Thus when random effects all add together, they often result in a normal distribution. Measurement error terms are typically modelled as normally distributed.

11.5.7 Exponential distribution

The *Exponential* distribution describes the time between rare events so always takes non-negative values. It has a single parameter, λ known as the rate and has density function

$$f(x) = \lambda e^{-\lambda x},$$

where $x \geq 0$. If X is exponentially distributed,

$$E[X] = \frac{1}{\lambda} \text{ and } \mathrm{Var}(X) = \frac{1}{\lambda^2}.$$

Write $X \sim Exp(\lambda)$.

11.5.8 Gamma distribution

The *Gamma* distribution arises as the sum of a number of exponentials. It has two parameters, k and θ, called the shape and scale, respectively. These parameters can be used to specify the mean and variance of the distribution.

$$f(x) = \frac{1}{\theta^k \Gamma(k)} x^{k-1} \exp(-x/\theta) \text{ for } x > 0,$$

where $\Gamma(k) = \int_0^\infty t^{k-1}e^{-t}\,dt$ is the gamma function (the extension of the factorial function, $k!$, to all real numbers). The mean and variance of a gamma distributed random variable X is

$$E[X] = k\theta \text{ and } \mathrm{Var}(X) = k\theta^2.$$

Write $X \sim Gamma(k, \theta)$.

Note that the gamma distribution has different parametrisations which result in different looking (but mathematically equivalent) expressions for the density, mean and variance — be sure to check which parametrisation is being used.

11.6 Entropy

Seeing this is a computer science course we should briefly discuss an idea in computer science which has deep connections to statistics: information theory. Central to information theory is the concept of *entropy*. Entropy measures the unpredictability associated with a (discrete) random variable. If X is a random variable with pdf $p(X)$, the Shannon entropy of X is

$$H(X) = -\sum_{x\in\mathcal{X}} p(x)\log(p(x)) = -E[\log(p(X))],$$

where we define $0\log 0 = 0$. The base of the logarithm used is arbitrary though is typically chosen to be 2, e or 10. When the base is 2, the units of entropy are *bits*, when it is e, the units are *nats*, and when it is 10, they are *dits*.

The entropy of X is maximised when p is the uniform distribution, that is $p(x) = 1/n$ for each of the n possible outcomes of X. Intuitively, this makes sense as in this case we are maximally uncertain about the outcome of X. At its maximum, $H(X) = \log n$. Conversely, the entropy is minimised taking value $H(X) = 0$ when $p(x) = 1$ for some x. In that case, there is no uncertainty about the outcome of X.

Example: What is the entropy of a fair coin toss?

Solution: Let X be the outcome of the coin toss. Then $p(x) = 1/2$ for either value of x. So, using log base 2, we get the entropy of X is $H(X) = -\sum_{x\in\mathcal{X}} p(x)\log(p(x)) = -(\frac{1}{2}\log(\frac{1}{2}) + \frac{1}{2}\log(\frac{1}{2})) = -\log(\frac{1}{2}) = 1$ bit. □

If the outcome of a random variable is known, the entropy (uncertainty) is reduced to zero. For this reason, the *information* contained in a signal can be thought of the difference in entropy before the signal was received to that after, that is, $I(X) = H_{\text{before}} - H_{\text{after}}$. If the signal is perfect so that the outcome of the random variable is completely known after receiving the signal, the initial entropy and information content are equivalent. If the signal is noisy, H_{after} is not zero and the original entropy is greater than the information content.

If we have two distributions, P and Q, we define the relative entropy (or Kullback-Leibler divergence) by

$$H(P||Q) = \sum_{x\in\mathcal{X}} P(x)\log\left(\frac{P(x)}{Q(x)}\right).$$

This quantity can be interpreted as the extra information required to code samples from P using a code based on Q. Sometimes you may see this measure described as a distance between two distributions, however $H(P||Q) \neq H(Q||P)$ so it not a distance (metric) in the mathematical sense. The KL-divergence sometimes appears in Bayesian statistics to measure the information gained by observing the data. In this case, we set P to be the posterior distribution (the post-data distribution) and Q to be the prior distribution (the pre-data distribution. Ideally, our experiments will be designed to maximise the KL-divergence between the prior and posterior.

12 Inference

Let's consider how we pose and tackle problems in a statistical framework. Suppose we have a statistical model for some real process (from biology, physics, sociology, psychology etc...). By having a model, we mean that given a set of control parameters, θ, we can predict the outcome of the system, D. For example, our model may be that each element of D is a draw from one of the distributions we described above so the control parameters θ are just the parameter(s) of that distribution. Note that the model of the process may include our (imperfect or incomplete) method of measuring the outcome.

In an abstract sense, then, we can consider the model as a black box with input vector θ (the parameters) and output vector D, the data.

$$\begin{tikzcd}
\begin{array}{c}\text{Parameter vector}\\ \theta\end{array} \arrow[r] & \boxed{\text{Model of system}} \arrow[r] & \begin{array}{c}\text{Data}\\ \text{D}\end{array}
\end{tikzcd}$$

The model gives us the forward probability density of the outcome given the parameter, that is, $P(D|\theta)$. This density is the called the likelihood, although, as we see below, we don't usually consider it as a density in the usual way.

This model is not deterministic. The data D can be seen as a random sample from the probability distribution defined by the model (and parameters). Changing the value of the parameters typically does not change the possible outcomes of the model but it will change the shape of the probability distribution, making some outcomes more likely, others less likely.

Example: Suppose we are interested the number of buses stopping at a bus stop over the course of an hour. We watch for the hour between 8am and 9am every weekday morning for 2 weeks. We observe the outcomes $D = (10, 7, 5, 6, 12, 9, 10, 5, 14, 7)$. A sensible model here might be the Poisson distribution where we say that the number of bus arrivals in an interval is Poisson distributed with parameter λ. Our parameter vector contains just the single parameter $\theta = (\lambda)$ and our data vector contains the 10 observed outcomes $D = (D_1, D_2, \ldots, D_{10}) = (10, 7, 5, 6, 12, 9, 10, 5, 14, 7)$.

We derive the likelihood as follows.

The probability of observing the data D for a given value of λ is $P(D|\lambda)$. Let's assume that each observation is independent of others then $P(D|\lambda) = \prod_i P(D_i|\lambda)$. That is, the probability of observing this series of outcomes is just the product of the probabilities of observing each particular outcome.

The likelihood of a single observation is given by the probability distribution function for the Poisson since $D_i \sim Poiss(\lambda)$ so:

$$P(D_i|\lambda) = \frac{\lambda^{D_i}}{D_i!} e^{-\lambda}.$$

And so the likelihood of observing the full data D is just

$$P(D|\lambda) = \prod_i P(D_i|\lambda) = \prod_i \frac{\lambda^{D_i}}{D_i!} e^{-\lambda}.$$

□

Note that the likelihood is a probability density function for D. But D is typically fixed in the sense that we make the observations which remain fixed through-out the analysis. We will be interested in considering the likelihood as a function of the parameters θ. The likelihood is *not* a probability density function for θ since, in general $\int_{\theta \in \Theta} P(D|\theta)\, d\theta \neq 1$.

12.1 Bayesian inference

The statistical problem essentially comes down to one of observing the outcome, D and wanting to recover the parameters θ.

That is, we want to estimate θ given D. We summarise our estimate of θ as a probability distribution, conditional on having observed D: $P(\theta|D)$. This is called the **posterior distribution** of θ.

From Bayes' theorem, we can express the posterior in terms of the likelihood:

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)},$$

where $P(D|\theta)$ is the **likelihood**, $P(\theta)$ is the **prior distribution** of θ and $P(D)$ is a normalisation constant.

The prior $p(\theta)$ summarises what we know about a parameter before making any observations.

The posterior, $p(\theta|D)$ summarises what we know about θ after observing the data.

The likelihood tells us about the likeliness of the data under the model for any value of θ. Recall that we consider the likelihood a function of θ rather than a probability density for D; to emphasise this fact, people often write it explicitly as a function of θ: $L(\theta) = P(D|\theta)$.

Bayes' theorem tells us how we update our beliefs given new data. Our updated beliefs about θ are encapsulated in the posterior, while are initial beliefs are encapsulated in the prior. Bayes' theorem simply tells us that that we obtain the posterior by multiplying the prior by the likelihood (and dividing by $P(D)$ which we can think of as a normalisation constant).

Note that we need the normalisation constant as the posterior is a probability distribution for θ, so its density must integrate to 1, i.e., $\int_{\theta\in\Theta} f(D|\theta)\,d\theta = 1$. Thus the normalisation constant is $P(D) = \int_{\theta\in\Theta} P(D|\theta)P(\theta)\,d\theta$. Typically this integral is hard to calculate so we try to find that will avoid having to calculate it.

Example: In the example above, we found an expression for the likelihood. To find an expression for the posterior, we need to decide on a prior distribution. Suppose we had looked up general info about bus stops in the city and found that the busiest stop had an average of 30 buses an hour while the quietest had an average of less than 1 bus per hour. We use this prior information to say that any rate parameter λ producing an average of between 0 ($\lambda = 0$) and 30 ($\lambda = 30$) buses an hour is equally likely. This leads us to the prior $\lambda \sim U(0, 30)$. The density of this prior is $f(\lambda) = 1/30$ for $0 \leq \lambda \leq 30$.

To get the posterior density, we use the formula above:

$$f(\lambda|D) = \frac{f(D|\lambda)f(\lambda)}{P(D)} = \frac{\prod_i \frac{\lambda^{D_i}}{D_i!} e^{-\lambda} \frac{1}{30}}{P(D)}.$$

The normalisation constant $P(D)$ is the integral of the numerator over all possible values of λ:

$$P(D) = \int_0^{30} \prod_i \frac{\lambda^{D_i}}{D_i!} e^{-\lambda} \frac{1}{30}\,d\lambda.$$

□

While it is possible to calculate this particular integral analytically, for most posterior distributions analytical integration is either very difficult or impossible. We'll investigate methods for avoiding calculating difficult integrals like this in later sections.

12.2 Maximum likelihood

It is often difficult or inconvenient to deal with the posterior distribution (when the prior is hard to specify or the normalisation constant is impossible to calculate). In these cases, we can still use our probabilistic model by concentrating solely on the likelihood function. The aim here is typically to find the parameters that maximise the likelihood function. That is, those parameters under which the observed data is most likely. We call this parameter estimate the maximum likelihood estimate and write it as

$$\hat{\theta} = \arg\max_{\theta} f(D|\theta) = \arg\max_{\theta} L(\theta; D)$$

This function can be maximise using standard tools from calculus (taking the derivative and setting it to zero – it is often easier to work with the log of the likelihood function as they both share a maximum) or using numerical techniques such as hill-climbing.

Many methods in statistics are based on maximum likelihood including regression, χ^2-tests, $t-$tests, ANOVA and so on.

Example: In the bus example above, we could find the maximum likelihood estimator for λ by differentiating the log-likelihood, $\log(L(\lambda; D))$ with respect to λ, setting the result to zero and solving. Note that we often work with the log-likelihood rather than the likelihood for a couple of reasons: it is often easier algebraically and it helps avoid numerical under-flow when the likelihood itself is very small.

13 Simulation

In a statistical setting there are a number of reasons we may wish to simulate from a distribution or a stochastic process. We may wish to get a feeling for how the process behaves or estimate some quantity that we cannot calculate analytically. An example of the latter case arises in Bayesian statistics, where the aim is to find the posterior distribution $f(\theta|x)$. This can be very difficult for two main reasons:

- The normalising constant ($P(D)$, above) involves a $p-$dimensional integral (where p is the number of parameters of the model, that is, $\theta = (\theta_1, \theta_2, \ldots, \theta_p)$) which is often impossible to calculate analytically.
- Even if we are able to find $f(\theta|x)$, if we want to find the marginal distribution for some part of θ this may again involve a high-dimensional integral.

Both of these reasons boil down to the fact that integration is hard.

To get around this problem, our approach will be to obtain a sample of values, $\{\theta^{(i)}\}$ for $i = 1, \ldots, n$, from the distribution of interest and use this sample to estimate properties of the distribution.

For example, the mean of the distribution is $E[\theta] = \sum_{\theta\in\Omega} \theta \Pr(\theta)$. We can estimate this from the sample set by $E[\theta] \approx \bar{\theta} = \frac{1}{n}\sum_1^n \theta^{(i)}$. This is called the sample mean of θ, and is indicated by the bar over the variable. The sample mean is an *estimator* of the mean. More generally, we estimate the mean of a function of θ by $E[g(\theta)] \approx \bar{g}(\theta) = \frac{1}{n}\sum_1^n g(\theta^{(i)})$.

How good are these estimates? Since each sample $\theta^{(i)}$ is a random variable, $\bar{g}$ is a random variable. That is, each time we obtain a different sample of values of θ, we will get a different value for $\bar{g}$. Clearly, as n, the size of the sample, increases our estimate will become more accurate but by how much?

It turns out that under quite general conditions, the main being that the samples, $\theta^{(i)}$ are independent of each other, $\bar{g}$ is normally distributed with mean $E[g]$ and variance $\text{var}(\bar{g}) = \text{var}(g)/n$. When stated formally, this is known as a central limit theorem.

Thus, if we have a method of simulating lots of independent samples, we can quickly get extremely accurate estimates of the quantities we are interested in.

Note that we can estimate more complex things than simple means using these methods. For example, we can estimate the shape of distributions by drawing a histogram of the sampled points.

So our attention turns to how we can generate this random sample. First we consider how we can generate or simulate randomness at all using (deterministic) computers.

13.1 Random number generation

All simulation relies on a ready supply of random numbers. There are currently no known methods to generate truly random numbers with a computer without measuring some physical process. There are, however, many fast and efficient methods for generating pseudo-random numbers that, for most applications, are completely sufficient. The fact that these are based on algorithms that are repeatable makes them superior to physically based rngs for scientific simulation purposes.

We do not go into the mathematical details of pseudo-random number generators here as most major languages have libraries that implement perfectly adequate algorithms. It is worth considering briefly what we want in a RNG. The following quality criteria are taken from L'Ecuyer in the Handbook of Computational Statistics, 2004.

The RNG must:

- have a very long period so that no repetitions of the cycle occur;
- be efficient in memory and speed;
- repeatable so that simulations can be reproduced;
- portable between implementations;
- have efficient jump-ahead methods so that a number of virtual streams of random numbers can be created from a single stream for use in parallel computations; and,
- have good statistical properties approximating the uniform distribution on $[0, 1]$.

It is relatively simple to come up with rngs that satisfy the first of these criteria, yet the last is where the difficulties occur. The performance of rngs can be tested via the diehard test suite (or more recently, the dieharder suite). See `http://www.phy.duke.edu/~rgb/General/dieharder.php`.

13.1.1 Linear congruential generators

The most basic rngs are probably the linear congruential generators that have the form $X_{n+1} = (aX_n + b) \bmod m$ where the constants a, b and m need to be chosen. We divide the number by m to get it in the range $[0, 1]$, that is, set $U_i = \frac{X_i}{m}$.

These have poor statical properties, however, and should be avoided for simulations.

13.1.2 Shift register generators

All numbers in computers are stored as a group of bits (32 bits or 64 bits). Shift register generators work directly with this representation to produce a sequence of random numbers. Start with a seed $U_0 = 0.b_{01}b_{02}\dots b_{0L}$ (where $L = 32$ or $L = 64$). Then get $U_i = 0.b_{i1}b_{i2}\dots b_{iL}$ by $b_{ij} = f_j(b_{(i-1)1}, b_{(i-1)2}, \dots, b_{(i-1)L})$ where f_j is some function $f : \{0, 1\}^L \to \{0, 1\}$.

Example: For $L = 4$, set $f_1 = b_{(i-1)3}\,\mathrm{XOR}\,b_{(i-1)4}$ and $f_j = b_{(i-1)(j-1)}$ otherwise. Starting with 0101 we get the following sequence:

```
0101
1010
1101
1110
1111
0111
0011
0001
1000
0100
0010
1001
1100
0110
1011
0101
```

$\square$

An rng that extends this idea is the so-called Mersenne Twister which is the statisticians rng of choice for simulation. Most languages have an implementation of this algorithm. See the wikipedia page `http://en.wikipedia.org/wiki/Mersenne_twister` for more details.

The Mersenne Twister is implemented in the Colt library for Java (see `https://acs.lbl.gov/software/colt/`). It is the default rng in the Python random library.

13.2 Simulating from univariate distributions via Inversion sampling

Simulation from discrete or continuous distributions with cumulative density function $F(X)$ relies on the following result which tells us that all we need to simulate draws from an arbitrary univariate distribution is a draw from $U(0,1)$ and use the inverse of the cdf:

Result (Inversion method): If $U \sim U(0,1)$, then $X = F^{-1}(U)$ produces a draw from X where F^{-1} is the inverse of X.

Thus when the cdf is known and we can find the inverse, sampling from the distribution is easy, as the following example shows.

Example (simulating an exponential random variable): if $X \sim \text{Exp}(\lambda)$, then $F(x) = \int_{-\infty}^{x} f(t)dt = \int_{-\infty}^{x} \lambda e^{-\lambda t}dt = 1 - e^{-\lambda x}$.

It is simple to see (by setting $u = 1-e^{-\lambda y}$ and solving for y) that $F^{-1}(u) = -\log(1-u)/\lambda$. Since $1 - U \sim U(0,1)$ when $U \sim U(0,1)$, we can use this expression to generate exponential random variables by generating the uniform random variable u and setting $x = -\log(u)/\lambda$. $\square$

Note that with discrete random variables, a inverse of F is ambiguously defined (since

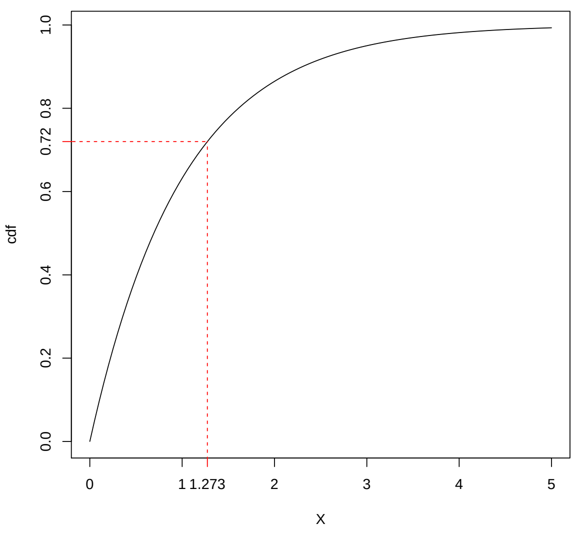


Figure 6: Sampling an $Exp(1)$ random variable using the inversion method. A uniform sample $u \sim U(0,1)$ is drawn. Here $u = 0.72$ shown on the vertical axis. This is mapped, via the cdf, to $x \sim Exp(1)$ to produce $x = -\log(1-u) = 1.272$ shown on the horizontal axis.

F is a step function). It is possible to extend the definition of an inverse to derive the following method for simulating discrete random variables.

Inversion sampling from a discrete distribution: If X is discrete with $P(X = x_i) = p_i$, we generate $U \sim U(0,1)$ and set $X = x_1$ if $u < p_1$ and $X = x_i$ if $\sum_{j=1}^{i-1} p_j < u < \sum_{j=1}^{i} p_j$.

Example: Use the inversion method to obtain samples from $X \sim Binomial$(n = 5, p = 0.3).

Solution: The possible values X can take are $(0,1,2,3,4,5)$ with probabilities $f(x) = (0.168, 0.360, 0.309, 0.132, 0.028, 0.002)$, respectively (from Section 11.5.3). Obtain the cdf by taking the cumulative sum of these probabilities: $F(X) = (0.168, 0.528, 0.837,$
$0.969, 0.998, 1.000)$. Now obtain samples from X by sampling $u \sim U(0,1)$ and finding the index of the smallest value of the cdf which is larger than u. E.g. if $u = 0.439$, $x = 1$ since $F(0) = 0.168 < u < F(1) = 0.528$, Similarly, if $u = 0.972$, $x = 4$ since $F(3) < u < F(4)$. $\square$

13.3 Stochastic processes

So far we have talked about random variables which can be pretty much any object but are only observed at one time. A stochastic process is a collection of random variables that describes the evolution of a system that is subject to randomness. A stochastic process could, for example, describe the position of a particle that is being buffeted by other particles, the state of a genetic sequence that is subject to copying with mutation, or the shape and size of a land mass that is subject to geological forces.

Mathematically, we consider a random process X as the set of random variables $\{X_t : t \in T\}$ where T is some index set, such as discrete time ($T = 0, 1, 2, \ldots$) or continuous time ($T = [0, \infty)$).

We will try to get an understanding of these process by studying a few examples.

13.3.1 Random walk

One of the simplest stochastic processes is known as the simple symmetric random walk, or drunkard's walk. Imagine a person leaves the pub so drunk that their method of getting home consists of taking random steps, with probability 0.5 the step is in the direction of home with probability 0.5 it is in the other opposite direction. We can model the drunk's position after the ith step as a random variable X_i where $X_0 = 0$ (that is, the pub is the origin). Then $X_{i+1} = X_i + S_i$ where

$$S_i = \begin{cases} +1 & \text{with probability } 1/2 \\ -1 & \text{with probability } 1/2. \end{cases}$$

with is the direction of the ith step. Equivalently, $X_i = X_0 + \sum_{j=0}^{i-1} S_j$. The process, while amenable to analytic techniques, is extremely simple to simulate: we just need to be able to simulate Bernoulli random variables.

The random walk has many variations: instead of looking at a symmetric walk, consider $S_i = 1$ with probability p; we can consider the random walk in higher dimensions, choosing from 2^d possible directions in d dimensions; and choosing a different step size ($S_i = \pm c$, say).

The process has some very nice, and often surprising, properties. For example, the simple symmetric random walk crosses every point an infinite number of times (this is known as Gambler's ruin, as if X models the amount a gambler is winning when betting \$1 on toss of a coin, the gambler will certainly eventually lose all their money if they play for long enough against a casino with infinite resources).

Secondly, the random walk in d dimensions returns to the origin with probability 1 for $d = 1, 2$, but for $d \geq 3$, that probability is below 1 (about 0.34 in for $d = 3$, 0.19 for $d = 4$ etc.).

13.3.2 Poisson process

The Poisson process is a simple yet incredibly useful model for events that occur independently of each other and randomly in time (or space). It is commonly used to model events such as:

- Genetic mutations
- arrival times of customers
- radioactive decay
- occurrences of earthquakes
- industrial accidents
- errors in manufacturing processes (e.g. silicon wafers or cloth)

We will consider processes in time although the concepts extend readily to space (the last of the examples above could be spatial).

A Poisson process is sometimes called a *point process*. It is counts the number of events in the time interval $[0, t]$. Let $N(t)$ be the number of points in the interval, so that $N(t)$ is a counting process.

Define a *Poisson process with intensity* λ, where $\lambda > 0$ (also called the *rate*) to be a process $\mathbf{N} = \{N(t), t \geq 0\}$ that takes values in $S = \{0, 1, 2, \ldots\}$ that satisfies the following properties:

1. $N(0) = 0$ and if $s < t$ then $N(s) < N(t)$.
2. If $s < t$ then $N(t) - N(s)$ is the number of arrivals in $(s, t]$ which is independent of the number (and times) of the arrivals in $(0, s]$.
3. $$\Pr(N(t+h) = n+m | N(t) = n) = \begin{cases} \lambda h + o(h) & \text{if } m = 1 \\ o(h) & \text{if } m > 1 \\ 1 - \lambda h + o(h) & \text{if } m = 0. \end{cases}$$

Here the notation $o(h)$ indicates that, as h gets small the bit of the expression that is $o(h)$ disappears. A strict definition is that function f is $o(h)$ ('of order little oh of h') if

$$\lim_{h \to 0} \frac{f(h)}{h} = 0.$$

Examples: Check that $f(h) = h^2$ is $o(h)$ while $f(h) = h^{-\frac{1}{2}}$ is not. $\square$

The Poisson process is related to the Poisson distribution by the fact that $N(t)$ has a Poisson distribution with parameter λt so that

$$\Pr(N(t) = k) = \frac{(\lambda t)^k}{k!} \exp(-\lambda t)$$

for $k \in \{0, 1, 2, \ldots\}$.

Now look at the times between events in a Poisson process. Let T_i denote the time of the ith event of the process and $T_0 = 0$. Then the ith *inter-arrival time*, $X_i = T_{i+1} - T_i$, is exponentially distributed with parameter λ, that is $X_i \sim \text{Exp}(\lambda)$, $i = 1, 2, 3, \ldots$.

Poisson processes have some very nice properties.

Splitting: Let $\{N(t), t \geq 0\}$ be a Poisson process with rate λ. Suppose each event is of type i, for $i \in \{1, \ldots, k\}$ with probability p_i and suppose that this is independent of other events.

If we observe just the events of type i, they form a Poisson process with rate λp_i independently of the remaining types of events.

For example, if we look at the request for different types of data in a network or arrivals of different types of customer, we get the large Poission process separated out as multiple smaller (lower rate) Poisson processes.

Merging: The converse of splitting is merging: Let $N = \{N(t), t \geq 0\}$ be a Poisson process with rate λ and $M = \{M(t), t \geq 0\}$ be a Poisson process with rate μ independent of N. Then $L = \{L(t) = N(t) + M(t), t \geq 0\}$ is a Poisson process of rate $\lambda + \mu$.

Together, these results tell us how to model multiple Poisson processes using a single large process: Suppose we have n independent Poisson processes where process i has rate λ_i. Then the merged process has events of n different types. If we observe an event in the merged process, let p_i be the probability of the event being of type i. What is p_i? According to the splitting theorem, the type i process has rate $\lambda_i = \lambda p_i$ where $\sum_i p_i = 1$ so that $\lambda = \sum_i \lambda_i$. So p_i is given by

$$p_i = \frac{\lambda_i}{\lambda} = \frac{\lambda_i}{\lambda_1 + \ldots + \lambda_n} = \frac{\text{rate of type } i}{\text{total rate}}.$$

Example: We model arrivals at a bus stop as a Poisson process. Some people arriving are students and some are office workers. Students arrive at rate λ_1, office workers arrive at rate λ_2. Merging these processes tells us the total rate of arrivals is $\lambda_1 + \lambda_2$. The probability that any given arrival is a student is $\frac{\lambda_1}{\lambda_1+\lambda_2}$ while the probability that they are an office worker is $\frac{\lambda_2}{\lambda_1+\lambda_2}$. $\square$

14 Markov chains

We think of a random process as a sequence of random variables $\{X_t : t \in T\}$ where T is an index set. T can be thought of as time. If T is discrete, the process $X(t)$ is called a *discrete time random process* while if T is continuous, $X(t)$ is called a *continuous time random process*. The random walk example is an example of a discrete time process (each time unit corresponds to a single step in the process) while the Poisson process is a continuous time process (arrivals happen at any time). We will consider only discrete time processes until further notice.

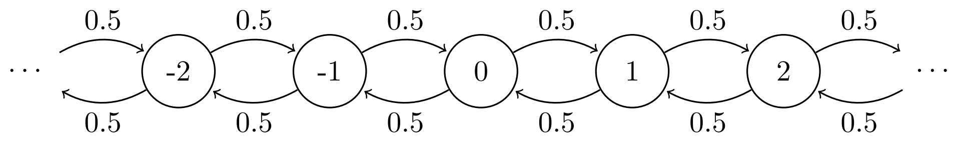
The random walk and Poisson processes described above both share an important property, known as the Markov property. Intuitively, this is the property of memorylessness in that future states depend only on the current state and not any past states. That is, to propagate the process forward, we need only be told the current state to generate the next state.

Formally, the sequence of random variables $X_1, X_2, X_3, \ldots$ is a *Markov chain* if it has the *Markov property*:

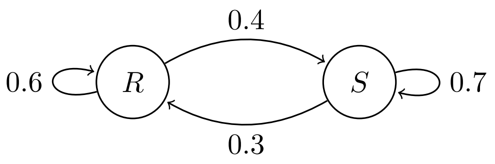
$$P(X_{n+1} = x | X_n = x_n, X_{n-1} = x_{n-1}, \ldots, X_2 = x_2, X_1 = x_1) = P(X_{n+1} = x | X_n = x_n).$$

Markov chains are commonly used to model processes that are sequential in nature and where the future state only depends on the current state. This limited dependence property is called the Markov property (after Andrey Andreyevich Markov, a Russian mathematician from the late 19th century).

Example 1: The random walk. Start at $X_0 = 0$. If X_n is the current state, $P(X_{n+1} = X_n + 1 | X_n) = 1/2 = P(X_{n+1} = X_n - 1 | X_n)$. This is a Markov chain on an infinite state space. A realisation of this chain: 0 1 0 -1 0 1 2 3 2 1 2 1 2 1 0 1



Example 2: Weather. The weather tomorrow depends on the weather today. If it is sunny today, tomorrow it is sunny with probability 0.7 and rainy otherwise. Rain clears a bit faster, so if it is rainy today, it is rainy tomorrow with probability 0.6 and sunny otherwise. This is a Markov chain with state space $\{R, S\}$ (for rainy and sunny, respectively). The following is a simulated realisation: S S S S S S R S R R R



Example 3: The following is **not** a Markov chain. Recall our random walk is called a drunkard's walk. Imagine someone occasionally helps the drunkard on the way home by carrying him 10 paces either to the left or the right. This person has limited patience, though, so will help at most 3 times. When the person has not yet reached the limit of their patience, possible transitions include $X_{n+1} = X_n \pm 10$ or $X_{n+1} = X_n \pm 1$. After the person has intervened to help 3 times, the only possible transitions are $X_{n+1} = X_n \pm 1$. So to see if this large movement is possible, we need to look back in history to see how many interventions have occurred. Thus the distribution of the next state depends on more than just the current state and the chain is not a Markov chain. □

The chain is *homogeneous* if $\Pr(X_{n+1} = j|X_n = i) = \Pr(X_1 = j|X_0 = i)$. If a chain is homogeneous, we write $P_{ij} = \Pr(X_1 = j|X_0 = i)$. The transition probabilities are normalised so that $\sum_j P_{ij} = 1$.

The matrix $P = [P_{ij}]$ is called a *stochastic matrix* as all its entries are non-negative and its rows sum to 1, so that $\sum_j P_{ij} = 1$.

Example: The transition matrix for the weather example given above is $\begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix}$ where rows and columns 1 and 2 are indexed by S and R, respectively. $\square$

A homogeneous Markov chain is completely defined by specifying an initial distribution $\Pr(X_0 = i)$ for X_0 and the transition probabilities X_{n+1} given X_n, P_{ij}.

Example 4: Music. See Figure 7. This example taken from Tom Collins, Robin Laney, Alistair Willis, and Paul H. Garthwaite. Chopin, mazurkas and Markov. Significance, 8(4):154-159, 2011. doi:10.1111/j.1740-9713.2011.00519.x.

Example 5: A DNA sequence. State space is $\{A, C, G, T\}$. Need to specify transition probabilities P_{AA}, P_{AC}, P_{AG} etc. Then we obtain a random sequence by specifying a starting state and recording each state visited. An example of a random sequence looks as follows: AAGCTGCGTGTGGGGGAAGGAACTTTTGCGTGTTAGTA

The $m-$*step transition probability* is the probability of going from state i to state j in exactly m steps, $P_{ij}(m) = \Pr(X_{n+m} = j|X_n = m)$. Hence the $m-$step transition matrix is $P_m = [P_{ij}(m)]$.

A result known as the *Chapman-Kolmogorov equations* tells us $P_{m+n} = P_m P_n$ (where the right-hand side is just standard matrix multiplication). In particular, this result tells us that $P_n = P^n$, that is, the n-step transition matrix is just the nth power of the (one-step) transition matrix.

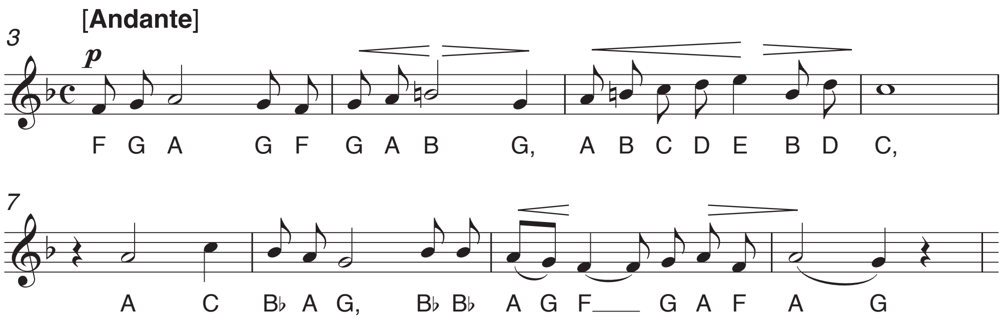

[Andante]
3
p
F G A G F G A B G, A B C D E B D C,
7
A C B♭ A G, B♭ B♭ A G F G A F A G

Figure 1. Bars 3–10 of the melody from "Lydia", Op. 4 No. 2, by Gabriel Fauré (1845–1924).

Table 1. Transition matrix for the material shown in Figure 1. The *i*th row and *j*th column records the number of transitions from the *i*th to the *j*th state in the melody, divided by the total number of transitions from the *i*th state.

Pitch class	*F*	*G*	*A*	*B♭*	*B*	*C*	*D*	*E*
F	0	3/4	1/4	0	0	0	0	0
G	2/7	0	4/7	1/7	0	0	0	0
A	1/8	1/2	0	0	1/4	1/8	0	0
B♭	0	0	2/3	1/3	0	0	0	0
B	0	1/3	0	0	0	1/3	1/3	0
C	0	0	1/3	1/3	0	0	1/3	0
D	0	0	0	0	0	1/2	0	1/2
E	0	0	0	0	1	0	0	0

1. A, G, F, G, F, G, A, B, G,
   F, G, F, G, A, B, D, E,
   B, C, A, F, G,
   B♭, A, F, G, A, G, A, B, G, A.

2. A, G, A, B, D, C, B♭, A, F,
   G, F, A, B, D, C, A, G,
   A, G, F, A, F,
   A, F, G, F, G, A, G, F, A, G.

3. F, A, B, G, F, G, F, G, A,
   B, C, A, G, F, G, F, G,
   B♭, A, G, A, G,
   A, F, G, B♭, A, B, G, F, G, A.

Figure 7: An example showing how a piece of music can be modelled as a Markov chain. The original piece, a fragment of Lydia by Fauré, is shown at the top. Just the pitches are considered in this simple Markov model. The transition matrix between pitches (centre) is constructed from empirical counts of the observed transitions. Three random realisations of the process are given at the bottom.

15 Introduction to genetics and genetic terminology

The history of life can be viewed, in a rather mundane way, as a long running and very complex stochastic (or random) process.

At a very basic level, and after many simplifying assumptions, we can think of the historical process explaining the relationships between species as a tree. The points where the tree splits are speciations and the leaves of the tree are different species. The past is back at the base or root of the tree and time increases from the root to the tips. Information is passed along the tree (away from the root) from one generation to the next via genetic material.

Genetic material is thought to be the only means by which biological information is passed from parent to offspring. The process of copying genetic material is imperfect, so that children will differ slightly from the parent. These imperfections consist of errors in the copying, known as mutations, and can be thought of as a stochastic process.

The fundamental objects we will be studying are sequences of characters representing biological macromolecules: DNA (Deoxyribonucleic), RNA (Ribonucleic acid) and proteins. DNA are RNA are the primary forms of genetic material. The characters in DNA and RNA sequences are drawn from 4 letter alphabets: DNA has $\Omega = \{A, C, G, T\}$ while RNA has $\Omega = \{A, C, G, U\}$. The A stands for adenine, C for cytosine, G for guanine, T for thymine and U for uracil. These are known as nucleobases or simply bases, with C, T, U being *pyramidines* and A, G being *purines*. Protein sequences consist of the twenty amino acids that are represented by the alphabet $\{A, R, N, D, C, E, Q, G, H, I, L, K, M, F, P, S, T, W, Y, V\}$ (that is, all the letters except $\{B, J, O, U, X, Z\}$). We will refer to the bases in an DNA/RNA sequence or the amino acids in a protein sequence as *residues*.

In eukaryotes (organisms with cells that have a nucleus), the three types of sequences related to each other by the Central Dogma of Molecular Biology that states, DNA makes RNA makes Protein. Or, more prosaically, DNA is *transcribed* into a type of RNA called mRNA that is then *translated* into protein.

There are some good animations showing how translation and transcription work at `www.hhmi.org/biointeractive/animations/index.html`, in particular see the DNA transcription and translation animations. A Japanese anime style film of the central dogma is also worth a look: `http://www.youtube.com/watch?v=-ygpqVr7_xs`.

Parts of the the DNA sequence encode information for proteins. These regions are known as genes and must be transcribed to RNA before being built into proteins. When the DNA is transcribed to RNA, all bases are copied exactly except that T (thymine) is transcribed as U (uracil). Once copied, the RNA is edited at splice sites so that only exons remain (the introns are edited out). This leaves the *messenger RNA*, mRNA, which is then translated to a protein sequence (poly-peptide chain). This translation occurs via the genetic code which translates consecutive triples of RNA bases (known as a codon) into one of the 20 amino acids. There are $4^3 = 64$ possible codes since there is an alphabet of 4 bases. 60 of these code for proteins, 1 (AUG) is a start codon and

3 (UAA, UGA and UAG) are stop codons signalling the start or finish of a protein. A particular amino acid may be encoded by just one codon (e.g. AUG→Methionine(M)) or up to 6 (e.g. any of UUA, UUG, CUU, CUC, CUA, CUG→Leucine (L)). Once the polypeptide chain is formed it folds into three dimensional molecule, taking on a particular structure.

Example: The sequence `atgaggttgacgctactttgttgcacctggagggaa` can be split into codons `atg agg ttg acg cta ctt tgt tgc acc tgg agg gaa` which translate into the protein sequence `MRLTLLCCTWRE`. □

In this course, we are only interested in the primary structure of sequences, that is, the order in which residues occur along the sequence. We will ignore the secondary, tertiary and quaternary structure of proteins — secondary structure is the name for the regular substructures such as alpha helices and beta sheets, the tertiary structure are the three dimensional structures of single molecules while quaternary structure are the complex forms taken by collections of single protein molecules. The study of these more complex structures is known as structural bioinformatics.

When DNA is passed from one generation to the next, the copy made is not exact. There are a number of processes that cause differences to arise between the parent and child. Recombination is one such process and involves the mixing of the maternal and paternal copies of DNA when the gametes (eggs or sperm) are produced. Other processes are generally thought of as mutations. The simplest are *point mutations* where the offspring sequence differs from the parent sequence by a single base (residue). This type of mutation is called a *single nucleotide polymorphism*, abbreviated as SNP and pronounced 'snip'. *Insertions* (or deletions) refer to the child sequence gaining (losing) one or more base than the parent. Larger scale mutations include: *gene duplication* which is a large scale insertion where the child inherits extra copy of a region containing a whole gene. Other large scale mutations include inversions (part of the sequence is reversed end to end) and translocations (a piece of the sequence is copied out of order).

Examples of mutations: Consider the short sequence `cgctcaccatgaagcgtttcactaat`. We demonstrate types of mutations showing the original sequence and a mutated version of it below with X marking the mutation.

- Single nucleotide polymorphism (SNP)
```
cgctcaccatgaagcgtttcactaat
cgctcgccatgaagcgtttcactaat
.....X....................
```
- Insertion
```
cgctcacc----atgaagcgtttcactaat
cgctcacctgatatgaagcgtttcactaat
........XXXX..................
```
- Deletion
```
cgctcaccatgaagcgtttcactaat
```

```
cgct----atgaagcgtttcactaat
....XXXX..................
```

- Duplication (the copied region is marked with parentheses). Note that duplication usually refers to gene duplication where whole genes are copied.

```
cgctcaccatgaagcgtttcacta-----------at
cgctcaccatgaagcgtttcactacaccatgaagcat
....(.........).........XXXXXXXXXXX..
```

- Inversion (again, this typically happens at a larger scale than shown here)

```
cgctcaccatgaagcgtttcactaat
cgctctaccagaagcgtttcactaat
.....XXXXX................
```

□

All these processes can be modelled and studied, with varying degrees of difficulty. We'll focus primarily on the question of how to align the sequences, how to identify regions of interest in sequences (for example, genes), and given aligned sequences, how can we reconstruct the evolutionary history (the tree) of those sequences. This last problem will require us to model the the mutation process where we restrict ourselves to looking at how point mutations arise.

The models we use will use are relatively simple, sometimes to the point of being downright crude. It is good to keep in mind the quote from the famous statistician George Box who said, "All models are wrong but some are useful".

15.1 Summary of above

- We model genetic sequences: think of them as strings of letters.
- There are 3 types of sequence, DNA, RNA or Protein.
- DNA sequences are composed of the 4 letters, or bases, $\{A, C, G, T\}$, RNA is made of the bases $\{A, C, G, U\}$ while protein sequences are made up of the 20 amino acids.
- The three types of sequence are related by the central dogma of molecular biology: DNA is transcribed to RNA and then translated to protein.
- Protein sequences fold up into more complex structures. We will ignore this structure in this introductory course.
- DNA is copied from parent to child.
- At copying, mutations are introduced.
- Mutations may be single nucleotide polymorphisms (SNPs), insertions, deletions or of other types.

- We use a tree to model the history of relationships between individuals (which are represented by their sequences).

To model the complex random process of genetic mutation and inheritance, we will need tools from applied probability and statistics. The next few sections are concerned with introducing the main tools and concepts that we will use for our study. All of you will have previously encountered at least some of the ideas we discuss here but, as with the linear algebra sections, it helps to review the main points before plunging in to new material.

16 Alignment

16.1 Homology

Homology (from the Greek, to agree) is a crucial concept in biology referring to traits or, in the case of sequences, sequence regions that share a common ancestry. We expect homologous regions to be similar to each other where the level of similarity will depend on how recently they shared a common ancestor.

Thus to say two regions are homologous is an evolutionary hypothesis. Mrs Darwin (in Carol Ann Duffy's poem from the collection The World's Wife) was making an evolutionary hypothesis of homology:

7 April 1852
Went to the Zoo.
I said to Him —
Something about that Chimpanzee over there reminds me of you.

The claim does not imply that the regions share a similar function now or, depending on the time since divergence, that they even particularly similar, just that they share a common ancestor. Therefore, sequences are either homologous or not, there are no degrees of homology. We often infer homology between two sequences when they are similar but we must be careful as we can get similarity without homology. Homologous sequences are sometimes referred to as *homologs*.

There are two main ways that we get similarity without homology: either by chance or by convergent evolution. Similarity by chance will occur even in completely random sequences on a finite alphabet. In two random sequences of four letters, we would expect similarity by chance of 25%.

Convergent evolution occurs when similar functions evolve independently of each other. An example of this are the wings of birds and insects. We don't believe these two very different creatures had a common ancestor that evolved wings but that wings evolved indecently of each other in the insect and bird lineages. Thus, while wings in a fly and a sparrow may be superficially similar, they are not considered homologous. The same applies to sequences that code for similar proteins (i.e., have similar function) but have evolved independently. Such traits/regions are called analogous.

We distinguish between two types of homology: orthology and paralogy:

Orthology occurs when two genes are separated by a speciation event and evolve independently from there on.

Paralogy occurs when a region of the genome is duplicated in the same genome (a duplication event) and they evolve in parallel in the same genome. The two copies are said to be paralogs.

16.2 Pairwise alignment

Given two sequences, if they are homologues, how do they align with each other? That is, exactly which sites in the sequence are homologous with each other?

We consider pairs of sequences, x and y of length m and n, respectively. x_i is the ith symbol of x. These symbols are usually the 4 DNA or RNA bases or the 20 amino acids. We refer to the symbols as *residues*.

We will allow *gaps* to be introduced in either sequence to allow them to align better. Biologically, gaps correspond to insertions or deletions in the sequence.

Clearly, there are many ways of aligning a pair of sequences (how many?), but what is the best alignment?

Example: x = GAATTC and y = GATTA

```
GAATTC or GAATTC-
GA-TTA    -GATT-A
```

are two possible alignments. □

16.3 Scoring alignments

The best alignment will depend on how we score alignments. It is easy to come up with different scoring regimes (e.g., score 1 for a match, -1 for a mismatch) but we really want to compare two models — that the similarity we see is just chance vs. that the sequences are homologs.

We initially consider alignments without gaps.

16.3.1 Model of non-homologous sequences

The most basic model is that each letter appears with some probability, letter a appears with probability q_a (note that the probabilities summed over the alphabet are 1), that each site is independent and that the each sequence is independent. Then the probability of seeing sequence x is

$$P(x) = \prod_{i=1}^{n} q_{x_i}$$

and the joint likelihood of an alignment is just the joint probability of the sequences x and y,

$$P(x, y) = P(x)P(y) = \prod_{i=1}^{n} q_{x_i} \prod_{i=1}^{m} q_{y_i} = \prod_{i=1}^{n} q_{x_i} q_{y_i}.$$

16.3.2 Model of homologous sequences

An alternative model is that the two sequences are related and the probability of seeing the pair of residues a (from x) and b (from y) aligned at a locus is p_{ab}. The probability of the alignment is then the product of the loci,

$$P(x, y) = \prod_{i=1}^{n} p_{x_i y_i}.$$

To compare these two models, we take ratio of these likelihoods:

$$\frac{\prod_{i=1}^{n} p_{x_i y_i}}{\prod_{i=1}^{n} q_{x_i} q_{y_i}} = \prod_{i=1}^{n} \frac{p_{x_i y_i}}{q_{x_i} q_{y_i}}.$$

It is easier to work with log-likelihoods (and addition) than likelihoods and multiplication, so we take the log of this quantity to get

$$S = \sum_i s(x,y_i)$$

where

$$s(a, b) = \log \left(\frac{p_{ab}}{q_a q_b} \right).$$

The score of the alignment then is the sum over the local score $s(a, b)$. s can be thought of as a matrix and is often referred to as a *score matrix* or *substitution matrix.*

There are various methods for deciding reasonable values for the entries of the matrix, discussed below. Note that even when ad hoc values are chosen for the matrix, the underlying probabilities, p_{ab} and q_a can be derived by reversing the above argument. That is, when a score matrix is selected we are making implicit assumptions about the q_as and p_{ab}s.

Example: If x = GAATTC and y = GGATTA are aligned as

```
GAATTC
GGATTA
```

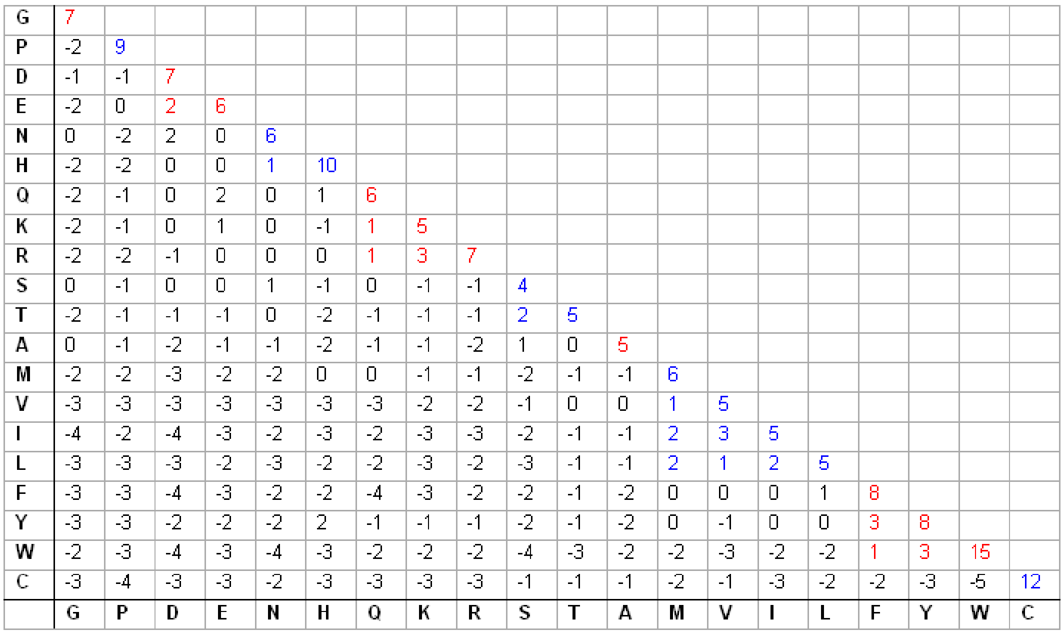
where $s(a, b) = 2$ if $a = b$ and $s(a, b) = -1$ if $a \neq b$, the alignments scores $2 - 1 + 2 + 2 + 2 - 1 = 6$. □

16.4 Choosing the substitution matrix

For protein sequences, the quantities p_{ab} and q_a have been empirically estimated to produce score matrices. In particular, the BLOSUM (BLocks SUbstitution Matrix) matrix of which there are various types, e.g. Blosum 45 and Blosum 80.

These matrices were calculated by studying a large number of confirmed alignments where there was considerable agreement between the sequences. The relative frequency of residues was calculated (to estimates for the q_as.) and the relative frequencies of pairs of residues was calculated (to give estimates for the p_{ab}s). The relative frequencies were then scaled to give integer entries in the matrix. The number after the matrix represents the similarity of the sequences used to estimate the matrix, so matrices with higher numbers are used for less divergent sequences.

The Blosum matrices are generally the most effective and widely used but see also PAM (Point Accepted Mutation) matrices.



G	7																			
P	-2	9																		
D	-1	-1	7																	
E	-2	0	2	6																
N	0	-2	2	0	6															
H	-2	-2	0	0	1	10														
Q	-2	-1	0	2	0	1	6													
K	-2	-1	0	1	0	-1	1	5												
R	-2	-2	-1	0	0	0	1	3	7											
S	0	-1	0	0	1	-1	0	-1	-1	4										
T	-2	-1	-1	-1	0	-2	-1	-1	-1	2	5									
A	0	-1	-2	-1	-1	-2	-1	-1	-2	1	0	5								
M	-2	-2	-3	-2	-2	0	0	-1	-1	-2	-1	-1	6							
V	-3	-3	-3	-3	-3	-3	-3	-2	-2	-1	0	0	1	5						
I	-4	-2	-4	-3	-2	-3	-2	-3	-3	-2	-1	-1	2	3	5					
L	-3	-3	-3	-2	-3	-2	-2	-3	-2	-3	-1	-1	2	1	2	5				
F	-3	-3	-4	-3	-2	-2	-4	-3	-2	-2	-1	-2	0	0	0	1	8			
Y	-3	-3	-2	-2	-2	2	-1	-1	-1	-2	-1	-2	0	-1	0	0	3	8		
W	-2	-3	-4	-3	-4	-3	-2	-2	-2	-4	-3	-2	-2	-3	-2	-2	1	3	15	
C	-3	-4	-3	-3	-2	-3	-3	-3	-3	-1	-1	-1	-2	-1	-3	-2	-2	-3	-5	12
	G	P	D	E	N	H	Q	K	R	S	T	A	M	V	I	L	F	Y	W	C

Figure 8: The Blosum 45 score matrix. The matrix is symmetric as $s(a,b) = s(b,a)$.

16.4.1 Scoring gaps

To make sequences align fully, we add gaps to one sequence or the other. A gap in x corresponds to an insertion in y with respect to x or a deletion in x with respect to y. Adding gaps comes with a penalty, so reduces the score for the match.

For a gap of length k, write $\gamma(k)$ for the penalty. We consider two forms for γ.

A *linear penalty* is defined by $\gamma(k) = -dk$ for some $d > 0$. That is, each deleted base adds a penalty of d.

An *affine penalty* is defined by $\gamma(k) = -d - (k-1)e$ where $d > e > 0$. d is the gap open penalty and e is the gap extension penalty. The affine penalty is more biologically appropriate as insertions or deletions are typically created in a single event rather than building up one residue at a time.

More complex gap penalties can be used, for example, we may wish to have different penalties for gaps matched with different residues, or non-linear functions of gap length. Such penalties come at the cost of more difficult implementation.

In the algorithms below, we'll first consider the simple case of the linear penalty.

16.5 Global alignment: Needleman-Wunsch algorithm

We can't calculate all possible alignments of a pair of sequences. (There are $\binom{n+m}{m}$ possible alignments for a pair of sequences of length n and m.) We use *dynamic programming* approaches that allow is to quickly calculate the best possible alignment (that is, the one that gives us the highest score).

Dynamic programming is technique of solving complex problems by reducing them to a number of much simpler subproblems that we can easily solve then re-assemble to find the answer to the complex problem. It uses the structure of the problem itself and so is only applicable to problems that possess a certain type of structure and to which we can apply the Principle of Optimality: "a sub-optimal solution of a sub-problem cannot be part of optimal solution of original problem".

The the alignment context, the principle of optimality holds in that if we know the score of an optimal alignment of length k then the score of the first $k-1$ parts of the alignment must be optimal.

To see why this is, let $F_{i,j}$ be the score of the optimal alignment between $x[1:i]$ and $y[1:j]$. Let $s(x_i, y_j)$ be the score for matching residue x_i to residue y_j and assume a linear gap penalty (so that the penalty for adding the gap $(x_i, -)$ or $(-, y_j)$ is d). The optimal alignment up to x_i, y_j either has x_i and y_j aligned, or y_j aligned to a gap or x_i aligned to a gap. For example, it looks like

$$\begin{array}{cccc} I & G & A & y_j \\ L & G & V & x_i \end{array} \text{ or } \begin{array}{cccc} I & G & A & y_j \\ V & x_i & - & - \end{array} \text{ or } \begin{array}{cccc} G & A & y_j & - \\ L & G & V & x_i \end{array}$$

In any case, the first part of the alignment must be optimal (If that first $k-1$ parts were not optimal, we'd find the optimal alignment for the first parts, add the kth bit on in one of the 3 possible ways and have a better alignment for the first k parts, contradicting our assumption that our original alignment was optimal for length k). Thus, if we know score of the best alignment for $k-1$ parts, we can extend it to the best alignment for k parts. This observation allows us to write the problem as a recurrence relation:

$$F(i,j) = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j), \\ F(i-1, j) - d, \\ F(i, j-1) - d. \end{cases}$$

The first case we have matched x_i, y_j, the second case we have matched x_i to a gap and the final case we have matched y_j to a gap.

So we want to find $F_{n,m}$ and we have the boundary conditions $F(0,0) = 0$ (start at 0), $F(i,0) = -id$ and $F(0,j) = -jd$ (linear gap penalties for initial gaps).

Note that all the above can be phrased in terms of mismatches and penalties, rather than matches and scores. To do so, simply reverse the signs of the scores and take minimums rather than maximums.

If we use a naive recursive method to calculate $F(n,m)$, we still get an exponential number of calls. But notice that there are only $m \times n$ possible combinations we need to calculate. We can do this in a tabular manner, calculating the matrix F from the top left to the bottom right in a progressive fashion.

To calculate the (i, j)th entry, we only need to know the 3 entries to the left and above it. The (i, j)th entry s then a maximum over 3 numbers. We keep a pointer to indicate which cell the (i, j)th entry was derived from.

$$\begin{tikzcd}
F(i-1,j-1) \arrow[dr, "+s(x_i,y_j)"'] & F(i-1,j) \arrow[d, "-d"] \\
F(i,j-1) \arrow[r, "-d"'] & F(i,j)
\end{tikzcd}$$

Once we have filled out the matrix, we trace back from $F(n, m)$, following the path that led us here. That is, the score at the (n, m)th position came from one of position $(n - 1, m)$, $(n, m - 1)$, or $(n - 1, m - 1)$ by adding a gap or a match. We move to whichever position it came from either adding the gap or the match in the process. In doing so, we build up the alignment from right to left, eventually arriving at $F(0, 0)$ at which point we can reverse the alignment to get the full

Our method is thus based on three things: a recurrence relation, tabular computing and then traceback. These methods turn an what is naively an exponential algorithm into a quadratic algorithm ($O(nm)$).

Example: Align `x = ATA` and `y = AGTTA` with the following scores: the purines are A and G, while the pyrimidines are C and T. Let $s(a, b) = 2$ if $a = b$, 1 if a is purine and b is a purine or a is a pyrimidine and b is a pyrimidine, and -2 if a is a purine and b is a pyrimidine or vice versa. Let the gap score be $d = -2$.

Solution: Filling out the matrix and drawing arrows to show where each entry is derived from we get the following:

$$\begin{tikzcd}
 & & \mathrm{A} & \mathrm{G} & \mathrm{T} & \mathrm{T} & \mathrm{A} \\
 & 0 & -2 \arrow[l] & -4 \arrow[l] & -6 \arrow[l] & -8 \arrow[l] & -10 \arrow[l] \\
\mathrm{A} & -2 \arrow[u] & 2 \arrow[ul] & 0 \arrow[l] & -2 \arrow[l] & -4 \arrow[l] & -6 \arrow[l] \arrow[ul] \\
\mathrm{T} & -4 \arrow[u] & 0 \arrow[u] & 0 \arrow[ul] & 2 \arrow[ul] & 0 \arrow[l] \arrow[ul] & -2 \arrow[l] \\
\mathrm{A} & -6 \arrow[u] & -2 \arrow[ul] \arrow[u] & 1 \arrow[ul] & 0 \arrow[u] & 0 \arrow[ul] & 2 \arrow[ul]
\end{tikzcd}$$

The score of the best alignment is given in the bottom right: $F(3, 5) = 2$. To find the alignment with the best score, we traceback from this point. At $F(2, 4)$ there are two choices that produce the same score. One alignment, found by following the arrow from

$F(2,4)$ to $F(1,3)$ is

$$\begin{array}{ccccc} A & - & - & T & A \\ A & G & T & T & A \end{array}$$

while the other is obtained by following the arrow from $F(2,4)$ to $F(2,3)$ and looks like

$$\begin{array}{ccccc} A & - & T & - & A \\ A & G & T & T & A \end{array}$$

□

16.6 Elements of an alignment algorithm

We emphasise that these dynamic programming algorithms for sequence alignment are based on following elements:

- a recurrence relation for the quantity we are trying to optimise, including specification of the boundary conditions,
- tabular computing to efficiently calculate the recurrence, and
- traceback (includes specifying rules for where to start and stop the traceback).

By altering the recurrence relation, the boundary conditions or the traceback, we will find different types of best alignment. Local alignment is the most common form and is defined below.

16.7 Local Alignment: Smith Waterman algorithm

The Needleman-Wunsch algorithm looks only at completely aligning two sequences. More commonly, we want to find the best alignment for some subsequence of two sequences. This is the local alignment problem.

The resulting algorithm that solves this problem is very similar to the one that solve the global alignment problem. We derive it as follows. Redefine $F(i,j)$ to be the score of the best suffix alignment of $x_1x_2\dots x_i$ and $y_1y_2\dots y_j$, where a *suffix alignment* is any alignment of $x_sx_{s+1}\dots x_i$ and $y_ry_{r+1}\dots y_i$ for some $1 \le s \le i$ and $1 \le r \le j$. Note that this suffix alignment could be the empty alignment with score 0.

We thus get the recursion

$$F_{i,j} = \max \begin{cases} 0 \\ F(i-1,j-1) + s(x_i,y_i), \\ F(i-1,j) - d, \\ F(i,j-1) - d. \end{cases}$$

So instead of a letting a score for an alignment to go negative, we start a new alignment. To find the best subsequence alignment, then, we simply look for the best score and trace it back until we hit a 0. Note than we can now start and finish anywhere in the matrix.

Example: Find the best local alignment using the score matrix and sequences given in the previous example: `x = ATA` and `y = AGTTA`

Solution: Fill out the matrix, drawing an arrow when a cell has a predecessor to get the following.

$$\begin{tikzcd}
 & & A & G & T & T & A \\
 & 0 & 0 & 0 & 0 & 0 & 0 \\
A & 0 & 2 \arrow[ul] & 1 \arrow[ul] & 0 & 0 & 2 \arrow[ul] \\
T & 0 & 0 \arrow[u] & 0 \arrow[ul] & 3 \arrow[ul] & 2 \arrow[ul] & 0 \arrow[u] \arrow[l] \\
A & 0 & 2 \arrow[ul] & 1 \arrow[ul] & 1 \arrow[u] & 1 \arrow[ul] & 4 \arrow[ul]
\end{tikzcd}$$

The score of the highest scoring local alignment is the largest entry in this matrix. We find this at $(5, 3)$ where $F(5, 3) = 4$. The sub alignment is found by tracing back from that cell and stopping at the first cell with no predecessor (or at the first 0 encountered). This produces the local alignment

$T \quad A$
$T \quad A$

□

16.7.1 Overlap matches

As an example of how easy it is to establish different types of alignment algorithm we consider a special type of alignment known as an overlap alignment.

When we expect one sequence to completely contain another or that they overlap, we want a global type alignment that does not penalize the unmatched overhanging ends. The boundary conditions are $F(i, 0) = F(0, j) = 0$ for all i, j, the recurrence relation is just the global recurrence and we start the traceback at the position on the right or top boundary the maximum score taken over all scores on those boundaries is achieved, $F(i, m)$ or $F(n, j)$. The traceback stops when either the left or top border is reached, $F(i, 0)$ or $F(0, j)$.

16.8 Pairwise alignment with non-linear gap penalties

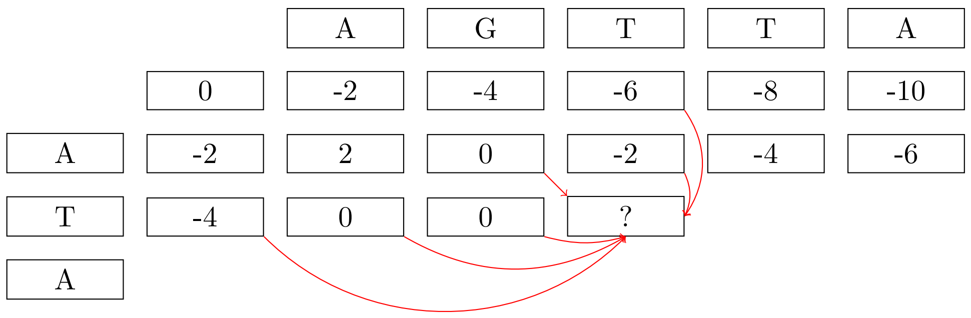
In our pairwise alignment discussion, we only considered linear gap penalties. As we noted earlier, linear penalties are a poor model for biological sequence data where we expect gaps (that is, insertions or deletions) to be quite rare but if there is a gap it may be multiple bases in length. Thus, an affine penalty, which penalises the start of the gap more heavily than any extension to the gap is favoured.

For an arbitrary gap penalty, $\gamma(k)$, we can continue to use a similar dynamic programming approach as before, but a direct adoption of that approach results in a much slower algorithm. Let's investigate: With a general gap penalty, $\gamma(k)$, the recurrence relation becomes

$$F_{i,j} = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j), & \\ F(k, j) + \gamma(i-k), & k = 0, \dots, i-1, \\ F(i, k) + \gamma(j-k), & k = 0, \dots, j-1. \end{cases}$$

This means that to calculate the value of each cell in the matrix $F(i, j)$ we need to consider $i + j + 1$ other cells — the i previous cells in the row, the j previous cells in the column, and the one adjacent diagonal cell — rather than the 3 as we had with the linear gap penalty (see Figure below). This results in a $O(n^3)$ algorithm rather than a $O(n^2)$

To calculate an unknown cell of F, the scores for gaps of all possible lengths need to be calculated meaning that a calculation for each previous cell in the row and column needs to be made:



16.9 Alignment with affine gap scores

In the case of an affine gap score (which has the form $\gamma(k) = -d - (k-1)e$) it turns out that we can, once again, construct a $O(n^2)$ dynamic programming algorithm to solve the alignment problem. The only difficulty is that we now have to keep track of 3 possible states corresponding to 3 cases:

1. Let $M(i, j)$ be the best score of the alignment up to (i, j) given that x_i is aligned to y_j. This case looks like
   ```
   A C C x_i
   A C G y_j
   ```

2. Let $I_x(i, j)$ be the best score of the alignment up to (i, j) given that x_i is aligned to a gap. This case looks like
   ```
   A C C x_i
   G y_j - -
   ```

3. Let $I_y(i,j)$ be the best score of the alignment up to (i,j) given that y_j is aligned to a gap. This case looks like
```
C C x_i -
A C G y_j
```

Given those definitions, and assuming that a gap cannot directly follow an insertion (that is, we can't go directly from I_x to I_y or vice versa), we have the following recurrence relations:

$$M(i,j) = \max \begin{cases} M(i-1,j-1) + s(x_i,y_j), \\ I_x(i-1,j-1) + s(x_i,y_j), \\ I_y(i-1,j-1) + s(x_i,y_j); \end{cases}$$

$$I_x(i,j) = \max \begin{cases} M(i-1,j) - d, \\ I_x(i-1,j) - e; \end{cases} \quad \text{and,}$$

$$I_y(i,j) = \max \begin{cases} M(i,j-1) - d, \\ I_y(i,j-1) - e. \end{cases}$$

It should be clear that we can calculate this recursion efficiently using tabular computation where we have 3 arrays: one for each of M, I_x and I_y. We can use a similar back-tracking mechanism to find the best alignment once we have calculated the scores.

This results in an quadratic time and space algorithm once again but the coefficient of the quadratic term is greater for this algorithm than the linear gap penalty one. For example, the space requirement here is $3n^2$ while it is only n^2 with a linear gap penalty.

The above recursions can be very neatly represented as a *finite state automaton*, or FSA.

In an FSA, each of the three possibilities, match, insertion in x or insertion in y, corresponds to a state (drawn as circles).

The transitions each carry a score, as indicated next to the arrow.

The new value for the state variable at (i,j) is the maximum of the scores corresponding to the transitions coming into the state. Each transition score is given by is given by the value of the source state at the offsets specified by the $\delta(i,j)$ pair of the target state plus the specified score increment.

An alignment corresponds to a path through the states.

```
V  L  S  P  A  D  -  K
H  L  -  -  A  E  S  K

m  m  Ix Ix m  m  Iy m
```

These automata are known in computer science as Moore machines.

We've already seen one type of FSA: a Markov chain can be represented as a stochastic FSA. We'll look at another stochastic FSA, the hidden Markov model or HMM, shortly.

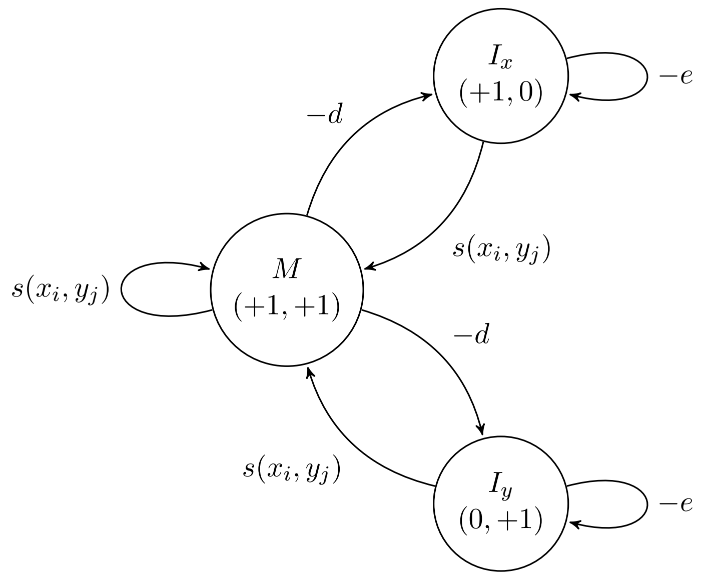


Figure 9: A finite state automaton describing the affine gap alignment recurrence relation. The pairs of numbers below the state names indicate how we increment the position in sequence x and y.

16.10 Linear space alignment

If sequences are large, even a quadratic algorithm can be difficult to work with. We can't improve the speed of the algorithm but we can reduce the amount of memory we need (currently at that is $O(n^2)$ too).

If all we require is the *score* of the best alignment, we immediately see that we don't need to keep the whole matrix until the end of the alignment. In the case of global alignment, the score of the best alignment is given by the entry $F(m, n)$. To calculate any score in the ith row, all we need to know is the $(i - 1)$th row, so we only need to keep the a single row of the matrix in memory. A similar argument can be made for the score of the best local alignment.

If we actually want the best alignment, it turns out that we can still produce a linear space algorithm. We employ a *divide and conquer* approach. Suppose we could find a cell, (i^*, j^*) what we knew to lie on the optimal alignment. Then we could divide the alignment problem into two halves: from $(0, 0)$ to (i^*, j^*) then from (i^*, j^*) to (m, n). In the best case, this reduces the amount of storage space require by 2. This process can be iterated, so (i^{**}, j^{**}) is found to reduce the space required for the $(0, 0)$ to (i^*, j^*) section and so on. It turns out that given an i^*, a suitable j^* can be found. We omit the details here (they are not too difficult) but details and references are given towards the end of chapter 2 in the Durbin et al book.

17 Multiple sequence alignments (MSA)

In pairwise alignment, we were given two sequences, x and y and wanted to align them by judiciously inserting gaps so that homologous residues were lined up with one another.

Multiple alignment is similar except now we have $k \geq 3$ sequences to align, so we want to form columns of homologous residues by adding gaps to each sequence.

Experts can construct multiple alignments by hand by considering a number of factors like secondary and tertiary protein structure, highly conserved regions, patterns of gaps, evolutionary processes etc. This is, however, subjective, difficult and tedious.

We want to come up with a method that is probabilistic, automatic and produces alignments that experts are happy with.

A good entry point to understanding the problem of global alignment is the `Phylo` puzzle game at `http://phylo.cs.mcgill.ca/` where you work on short multiple alignments by hand.

17.1 Dynamic programming

It is tempting to try to extend the dynamic programming methods that we used for pairwise alignments to the MSA problem. It is relatively easy to write down the naive extension of pairwise alignment algorithms to more than 2 sequences. But the naive implementation quickly becomes impossible as the number of sequences and the length of the sequences increases.

For example, for 2 sequences of length L, the Needleman-Wunsch algorithm requires a 2-dimensional array with L^2 cells in total to be stored in memory. The MSA analogue on n sequences requires storing an n-dimensional array containing L^n cells in total. Even for short sequences of $L = 100$, we would require memory for $100^5 = 10^{10}$ cells (at 4 bytes per cell, that's about 37 GB).

Some clever work from Lipman, Altschul and Kececioglu (the first two co-wrote BLAST) managed to reduce the size of the space that needs to be considered. That is, instead of calculating all L^n cells, they calculate upper and lower bounds on the score of the best MSA and then need only calculate the cells in the n-dimensional array that will produce scores lying between these two bounds. This work allows a few sequences (5-10) of moderate length (300 residues) and not too far diverged to be optimally aligned but even this requires a large computational resource.

17.2 Progressive alignment

Finding the optimal MSA is computationally prohibitive, as discussed in the previous Section. Typically, we resort to finding a good-enough alignment using heuristic techniques. The most widely used heuristic is progressive alignment.

Progressive alignment involves a series of successive pairwise alignments. At it's most basic, an initial pair of sequences are chosen and aligned, a third is chosen and aligned to the first two and so on until all sequences are included in the MSA. Other methods also allow the aligning of two alignments to each other. For example, if there are 4 sequences, two pairs may be aligned first then the two alignments aligned to complete the MSA.

These methods require that we can: decide on an order in which to align the sequences, align two sequences together, align a sequences to a MSA and align two MSAs together.

The typical way to decide on the order in which to align the sequences is to build a guide tree, using a clustering methods such as UPGMA, and align sequences in the order that nodes occur from the leaves to the root of the tree.

17.3 Building trees with distances and UPGMA

UPGMA is a method of building trees based on distances: we are given a set of objects, and for each pair of objects we have some measure of the distance between them.

UPGMA stands for unweighted pair group method using arithmetic averages and is a simple method with an ugly name. The idea can be thought of as a clustering algorithm where we start with all individual sequences and start clustering them together, building the tree up from the leaves to the root. The height of the internal nodes (or, equivalently, the edge lengths) is determined by the distances between the two clusters being joined.

For two sequences, x and y, we assume we have a method of defining the distance d_{xy}. We define the distance between two clusters of sequences, C_i and C_j as the average distance between all pairs between clusters:

$$d_{ij} = \frac{1}{|C_i||C_j|} \sum_{x \in C_i, y \in C_j} d_{xy}$$

where $|C|$ is the number of sequences in cluster C.

We define the algorithm as follows:

Initialise Assign each sequence i to it's own cluster C_i. Assign a leaf node to each cluster and give it height 0.

Repeat until there only one cluster remains Find clusters C_i and C_j such that d_{ij} is minimal (choose randomly between equidistant candidates).

Join i and j to make the new cluster $C_k = C_i \cup C_j$.

Define a node k in the tree placed at height $d_{ij}/2$ with child nodes i and j.

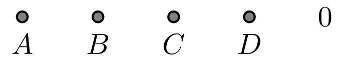
Update the distance matrix.

This procedure results in a well-defined tree (we need to check that all node heights are above the heights of the their children). The algorithm is quadratic ($O(n^2)$) in the number of sequences.

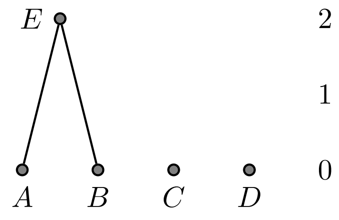
Example: Given 4 sequences, A, B, C and D, which have the pairwise distances given by the distance matrix d below, construct the UPGMA tree.

$$d = \begin{array}{c|cccc} & A & B & C & D \\ A & - & 4 & 8 & 8 \\ B & & - & 8 & 8 \\ C & & & - & 6 \\ D & & & & - \end{array}$$

Solution: Start by assigning leaf nodes to each of the sequences with height 0:



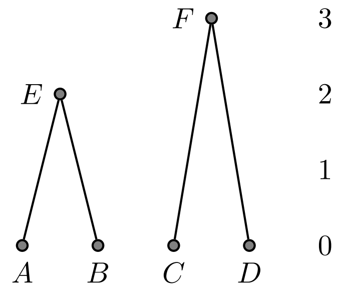
Now choose the pair of clusters that are closest to each other according to the distance matrix d. This is the pair (A, B) with distance $d(A, B) = 4$. Join the cluster $E = A \cup B = \{A, B\}$ which has height $d(A, B)/2 = 2$.



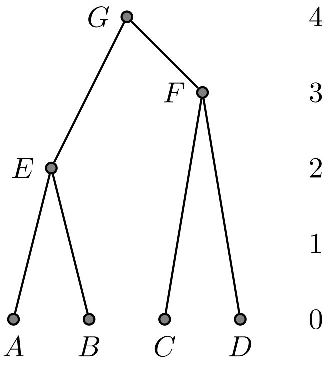
The distance matrix is by calculating $d(E, C)$ and $d(E, D)$. $d(E, C) = \frac{1}{2 \cdot 1}(d(A, C) + d(B, C)) = \frac{1}{2}(8 + 8) = 8$ and similarly for $d(E, D)$.

$$\begin{array}{c|ccc} & E & C & D \\ E & - & 8 & 8 \\ C & & - & 6 \\ D & & & - \end{array}$$

Now form the cluster $F = \{C, D\}$ and place the node at $d(C, D)/2 = 3$.



The distance matrix is now the single distance between the remaining clusters: $d(E, F) = \frac{1}{2 \cdot 2}(d(A, C) + d(A, D) + d(B, C) + d(B, D)) = \frac{1}{4}(8 + 8 + 8 + 8) = 8$. So make the last node $G = \{E, F\}$ and place it at height $8/2 = 4$. The UPGMA tree is thus



□

17.4 Feng-Doolittle progressive alignment

The Feng-Doolittle algorithm (1987) takes the approach described above. The steps for aligning n sequences are as follows.

1. Calculate the $n(n-1)/2$ distances between all sequences pairs. The distances are found by aligning each pair and recording a normalized score. The score used is

$$D = -\log S_{eff} = -\log \frac{S_{obs} - S_{rand}}{S_{max} - S_{rand}}$$

   where S_{obs} is the score from the pairwise alignment, S_{max} is the average of scores obtained by aligning each sequences of the pair to itself and S_{rand} is the expected score for an alignment of the pair when the residues are randomly shuffled. The effective score S_{eff} can thus be viewed as a normalised percentage similarity which roughly decays exponentially towards zero with increasing evolutionary distance. Thus, we take $-\log$ to make the score decay approximately linearly with evolutionary distance.

2. Build a guide tree based on the recorded scores (we use UPGMA).

3. Build the alignment in the order that nodes were added to the tree.

A pair of sequences is aligned in the normal way. A sequence is aligned to an MSA by aligning it to each sequence in the MSA and choosing the highest scoring alignment. Two alignments are aligned to each other by aligning all pairs of sequences between the two groups and choosing the best alignment.

After a sequence or group of sequences is added to an alignment, the introduced gap characters are replaced with a neutral X character which can be aligned to any other character (gap or residue) with no cost. Crucially, there is no penalty for aligning a gap to an X which tends to make gaps align with each other, giving us the characteristic pattern we see in multiple alignments of gaps clustered in columns.

Once the initial MSA has been found, we can use further heuristics to improve it and lessen any effect of the order in which sequences were added. For example, we can choose a sequence uniformly at random, remove it from the MSA and then realign it to the the MSA. This process can be iterated until the MSA becomes stable, that is, no or very few changes occur when a sequence is removed and re-aligned

18 Hidden Markov Models

In Markov models, we directly observe the state of the process (for example, in a random walk, the state is completely described by the position of the random walker). We can easily imagine a system where we don't directly observe the state but observe some outcome which depends on the state. Call the observed outcomes *symbols*. We thus make a distinction between the **sequence of states** and the **sequence of symbols**. We formally model these systems as hidden Markov models (HMMs).

Example: Weather. Suppose we live in a place that gets winds either from the south or north. On days the wind is from the south, it is rainy and cold 75% of the days, and warm and sunny the other 25% of the days. When the wind is from the north, is is rainy and cold 20% of days, and warm and sunny 80% of the days. If it is northerly today, it will be northerly tomorrow with probability 0.7. If it is southerly today it will be south with prob. 0.6. □

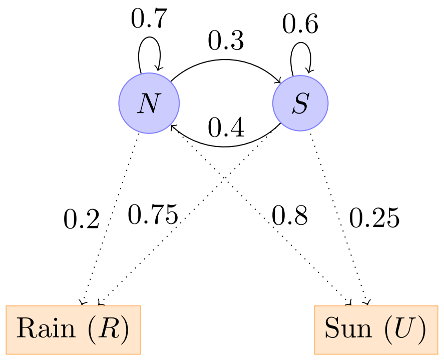


Figure 10: The chance of Sunny or Rainy weather on a day is determined by the wind direction. When it is Northerly, Sun is more likely, when it is Southerly, Rain is more likely.

The idea in the HMM is that we only observe the sequence of symbols but to understand what is going on, we need to know what the underlying state is.

Example cont.: A state sequence for the weather example might be $NNNSSNS$ while the emissions sequence looks like (using R for rain and U for sun) $UUUURUR$. We might observe the weather (rainy or sunny) and wonder what is causing the patterns we see. The pattern is best understood by knowing the sequence of states (direction of wind). □

Notation for HMMs

The ith state of the state sequence is π_i and transitions (of the states) is given by $a_{kl} = P(\pi_i = l | \pi_{i-1} = k)$. We use b's to represent the symbols so we get a new set of parameters called emission probabilities, $e_\pi(b) = P(x_i = b|\pi_i = \pi)$ is the prob of emitting symbol b given that we are in state π.

Example cont.: In the weather example $a_{NN} = 0.7$, $a_{NS} = 0.3$, $a_{SS} = 0.6$ and $a_{NS} = 0.4$. The emission probabilities are $e_N(R) = 0.2$, $e_N(U) = 0.8$, $e_S(R) = 0.75$ and $e_S(U) = 0.25$. □

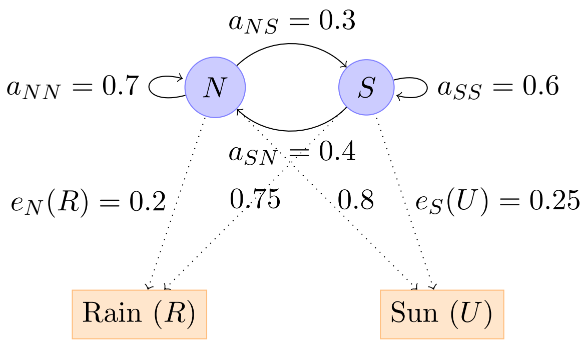


Figure 11: The chance of Sunny or Rainy weather on a day is determined by the wind direction. When it is Northerly, Sun is more likely, when it is Southerly, Rain is more likely.

Example: Cheating Casino. A casino tips things in its favour at a dice table by occasionally switching from a fair die to a loaded one. The loaded die produces a 6 50% of the time, and each of 1-5 just 10% of the time. Call the fair die F and the loaded die L. We have transitions $a_{FL} = 0.05$ and $a_{LF} = 0.1$. $a_{FF} = 1 - a_{FL}$ and $a_{LL} = 1 - a_{LF}$. Emissions are $e_F(b) = 1/6$ for $b = 1, \dots, 6$ while $e_L(b) = 1/10$ for $b = 1, \dots, 5$ and $e_L(6) = 0.5$.

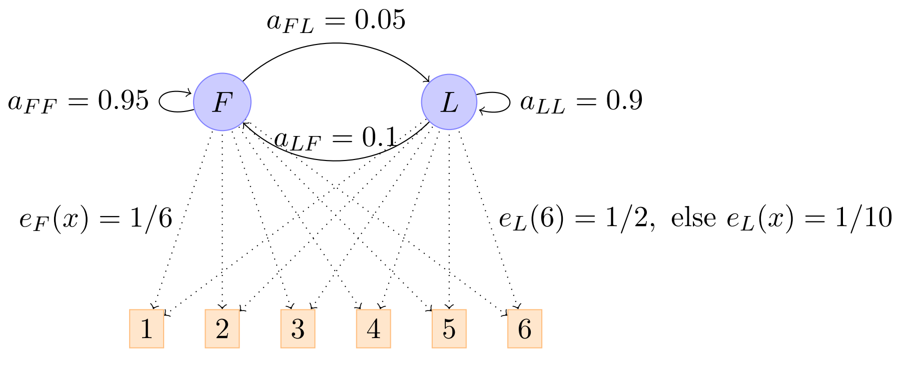


Figure 12:

A run of the chain might look like:

```
F F F F L F F F L L L L L L L L F F F F F F F F F F F F F F L L L L F F F
```

```
F L L L L L L F F F F F F F F
```
producing rolls
```
2 5 1 6 6 2 4 2 4 5 6 6 3 6 2 4 2 2 3 4 6 3 6 5 3 4 5 2 3 5 1 6 6 6 6 2 4
6 6 2 6 6 6 6 6 1 5 1 6 4 1 2.
```
□

Think of HMMs as a generative process where the first state π_1 is chosen according to probability a_{0i}. From that state, a symbol is emitted according to the probabilities e_{π_1}. State 2 is then chosen according to transition probs $a_{\pi_1 i}$ which emits a symbol and so on.

The joint probability of states and symbols is then,

$$P(x,\pi) = a_{0\pi_1} \prod_{i=1}^{L} e_{\pi_i}(x_i) a_{\pi_i \pi_{i+1}}$$

18.1 The Viterbi algorithm for finding the most probable state path

Multiple underlying states could produce the same path. Clearly, though, some states are more likely than others given the emissions. If we see lots of very good weather, for example, we may guess that northerlies are predominating in our city. If we see long strings of 6s, we might guess that the casino has switched to the loaded die, while sequences of apparently evenly distributed throws suggest a fair die is being used.

If we want to work with just one state path, it can be argued that the state path with the highest probability is the one to choose. Let

$$\pi^* = \arg\max_{\pi} P(x,\pi).$$

We can find π^* using a recursive algorithm due to Andrew Viterbi (an Jewish Italian refugee from the fascists who grew up in the USA, wrote this algorithm while at MIT and then started Qualcomm, a now huge technology company).

Let $v_k(i)$ be the probability of the most probable state path ending in state k at observation i (that is, considering only the first i observations). Suppose $v_k(i)$ is known for all states k. Then the next probability $v_l(i+1)$ can be calculated for observation x_{i+1} as

$$v_l(i+1) = e_l(x_{i+1}) \max_k (v_k(i) a_{kl}).$$

If we insist that all sequences start in a begin state called 0, we have $v_0(0) = 1$ and we can use a backtracking procedure like we saw in the pairwise alignment algorithm to find the most probable path.

The Viterbi algorithm thus proceeds as follows:

Initialisation $i = 0$: $v_0(0) = 1$, $v_k(0) = 0$ for $k > 0$.

Recursion $i = 1, \ldots, L$: $v_l(i) = e_l(x_i) \max_k (v_k(i-1) a_{kl})$.
$\quad Pointer_i(l) = \arg\max_k (v_k(i-1) a_{kl})$

Termination: $P(x, \pi^*) = \max_k(v_k(L)a_{k0})$. $\pi^*_L = \arg\max_k(v_k(L)a_{k0})$.

Traceback $i = L, \ldots, 1$**:** $\pi^*_{i-1} = Pointer_i(\pi^*_i)$.

Here we assume an end state is modeled, as a_{k0}. If the end is not modeled, this term disappears.

Even the most probable path typically has a very low likelihood, so it is important to work in log units. The recursion in log units is

$$V_l(i+1) = \log(e_l(x_{i+1})) + \max_k(V_k(i) + \log(a_{kl}))$$

where $V_k(i) = \log(v_k(i))$.

Example: In the human genome, the dinucleotide (that is, two base pairs next to each other in the sequence) CG, written CpG to distinguish it from the nucleotide pair C-G, is subject to methylation. Methylation changes the G to a T. That means we see the CpG dinucleotide less frequently than we would expect by considering the individual frequencies of C and G. In some functional regions of the genome, such as promoter and start regions of genes, methylation is suppressed and CpG occurs at a higher frequency than elsewhere. These are called CpG islands.

To detect these regions we model each region of a sequence as having either high or low CG content. We label the regions H and L. In high CG regions, nucleotides C and G each occur with probability 0.35 and nucleotides T and A with probability 0.15. In low CG regions, C and G are probability 0.2 and T, A are 0.3. The initial state is H with probability 0.5 and L with probability 0.5. Transitions are given by $a_{HH} = a_{HL} = 0.5, a_{LL} = 0.6, a_{LH} = 0.4$. Use the Viterbi algorithm to find the most likely sequence staes given the sequence of symbols $x = GGCACTGAA$.

Solution: We work in the log space (base 2) so that the numbers don't get too small. We'll need the log values of each of the above probabilities which are best represented as matrices: set $\mathbf{A} = \log_2(a)$ where a is the transition matrix

$$\mathbf{A} = \begin{array}{ccc} & H & L \\ H & \log(a_{HH}) = -1 & -1 \\ L & -1.322 & -0.737 \end{array}$$

and if $\mathbf{E} = \log_2(e)$ where e is the matrix if emission probabilities then

$$\mathbf{E} = \begin{array}{ccccc} & A & C & G & T \\ H & \log(e_H(A)) = -2.737 & -1.515 & -1.515 & -2.737 \\ L & -1.737 & -2.322 & -2.322 & -1.737 \end{array}$$

The matrix V, with row indices k and column indices i (so that the (k,i)th element is $V_k(i)$) along with the pointers to allow traceback is

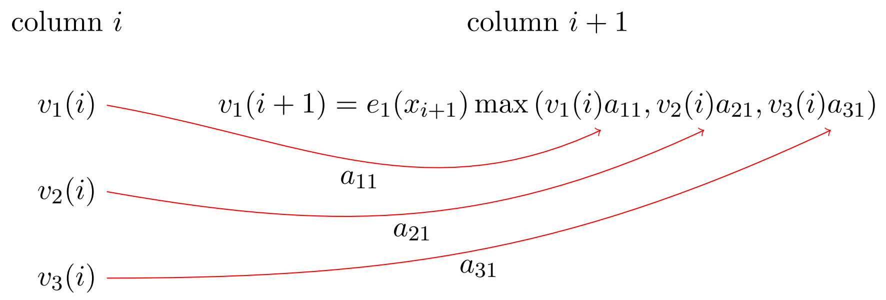
$$\begin{array}{cccccccccccccccccccc} & - & & G & & G & & C & & A & & C & & T & & G & & A & & A \\ 0 & 0 & & & & & & & & & & & & & & & & & & \\ H & -\infty & \nwarrow & -2.51 & \leftarrow & -5.03 & \leftarrow & -7.54 & \leftarrow & -11.28 & \swarrow & -13.12 & \leftarrow & -16.85 & \swarrow & -18.65 & \leftarrow & -22.39 & \swarrow & -25.41 \\ L & -\infty & \nwarrow & -3.32 & \nwarrow & -5.84 & \nwarrow & -8.35 & \nwarrow & -10.28 & \leftarrow & -13.34 & \leftarrow & -15.81 & \leftarrow & -18.87 & \leftarrow & -21.35 & \leftarrow & -23.82 \end{array}$$

The first column in this matrix is simple: every sequence is in state 0 at step 0, so $V_0(0) = \log(1) = 0$ while other states have $V_H(0) = V_L(0) = \log(0) = -\infty$.

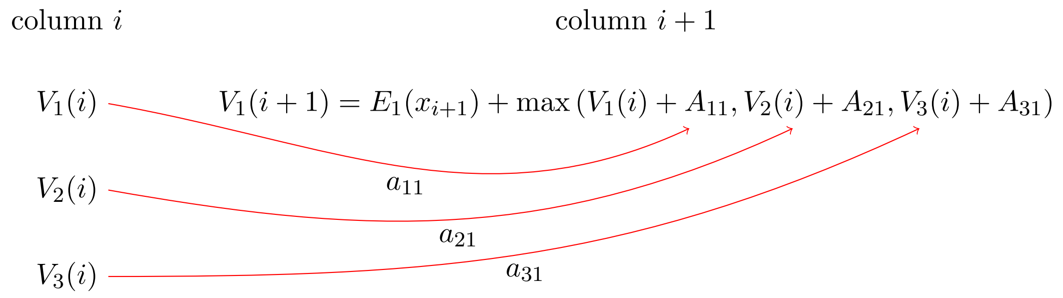
The second column is derived from the first as follows:
$V_H(1) = \log(e_H(G)) + \max\{V_0(0) + \log(a_{0H}), V_H(0) + \log(a_{HH}), V_L(0) + \log(a_{LH})\} = -1.515 + (V_0(0) + log(a_{0H})) = -1.515 - 1 = -2.515$ and similarly for $V_L(1)$.

Traceback begins in the final column where we see the state that maximises the joint probability is L. Following the pointers from this position and recording the state at each step gives us the state path with the highest probability is $\pi^* = HHHLLLLLL$. Note that π^* is built from right to left in the traceback procedure. □

The schematic below shows how the $(i+1)$th column is derived from the ith column in a Viterbi matrix. Here, there are 3 possible states, 1, 2 and 3. The 0 state is omitted in this diagram.


column i
column $i+1$
$v_1(i)$
$v_1(i+1) = e_1(x_{i+1}) \max(v_1(i)a_{11}, v_2(i)a_{21}, v_3(i)a_{31})$
$v_2(i)$
$v_3(i)$
a_{11}
a_{21}
a_{31}

The same diagram shown using log units:


column i
column $i+1$
$V_1(i)$
$V_1(i+1) = E_1(x_{i+1}) + \max(V_1(i) + A_{11}, V_2(i) + A_{21}, V_3(i) + A_{31})$
$V_2(i)$
$V_3(i)$
a_{11}
a_{21}
a_{31}

18.2 The forward algorithm and calculating P(x)

We have seen that it is easy to calculate $P(x, \pi)$. However, we usually only observe x so can't directly calculate $P(x, \pi)$. We could tackle this by finding a suitable state path, such as the Viterbi path, π^*, and calculate $P(x, \pi^*)$. But calculating $P(x, \pi^*)$ does not adequately tell us the likelihood of observing x which may have arisen from a large number of possible state paths.

What we really want is calculate is $P(x)$, the probability of observing x without taking any particular state path into account. This involves marginalizing over all possible paths: that is,

$$P(x) = \sum_{\pi} P(x, \pi).$$

The number of possible state paths grows exponentially so we cannot enumerate them all and naively calculate this sum. Instead, we use another dynamic programming algorithm called the forward algorithm and calculate $P(x)$ iteratively.

The forward algorithm iteratively calculates the quantity

$$f_k(i) = P(x_{1:i}, \pi_i = k),$$

the joint probability of the first i observations and the prob that $\pi_i = k$. The recursion used is that

$$f_l(i+1) = e_l(x_{i+1}) \sum_k f_k(i) a_{kl}. \tag{6}$$

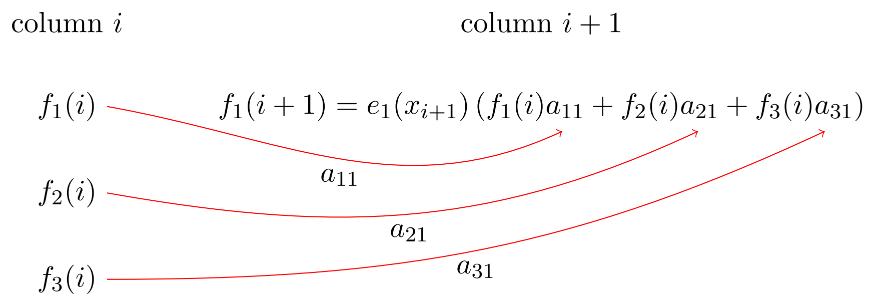
Initialisation $i = 0$**:** $f_0(0) = 1$, $f_k(0) = 0$ for $k > 0$.

Recursion $i = 1, \ldots, L$**:** $f_l(i) = e_l(x_i) \sum_k f_k(i-1) a_{kl})$.

Termination: $P(x) = \sum_k f_k(L) a_{k0}$.

If the end state 0 is not modelled, simply set $a_{k0} = 1$ to get $P(x) = \sum_k f_k(L)$.

Here's a diagram showing how to get the $(i+1)$th column from the ith column in the forward algorithm. The example shown has three states 1, 2 and 3.


column i
column $i+1$
$f_1(i)$
$f_1(i+1) = e_1(x_{i+1}) \left(f_1(i)a_{11} + f_2(i)a_{21} + f_3(i)a_{31}\right)$
a_{11}
$f_2(i)$
a_{21}
$f_3(i)$
a_{31}

Once again, we'll need to work with the log quantities as the qualities of interest get very small very fast. However, if we take the log of both sides of Equation 6, the log of the sum on the right hand side does not simplify immediately.

Let $F_k(i) = \log(f_k(i))$ and $A_{kl} = \log(a_{kl})$. Then Equation 6 becomes

$$F_l(i) = \log[e_l(x_i) \sum_k f_k(i-1) a_{kl})] = \log[e_l(x_i)] + \log\left[\sum_k \exp(F_k(i-1) + A_{kl})\right]$$

Directly calculating a sum of the form $\log(c) = \log(e^a + e^b)$ requires calculating e^a and e^b which we were trying to avoid all along. Instead, note that $\log(e^a + e^b) = \log(e^a(1 + e^{b-a})) = \log(e^a) + \log(1 + e^{b-a}) = a + \log(1 + e^{b-a})$. If the difference $b - a$ is not too large, this method never need store an extremely large or small number so is numerically stable. This extends to finding the log of a sum of multiple logged numbers:

```
function logsum(x)
  return x[0] + log(sum(exp(x - x[0])))
```

or, if you are using $\log_2$,

```
function log2sum(x)
  return x[0] + log2(sum(2^(x - x[0])))
```

Example cont: For the CG island example above, use the forward algorithm to calculate the probability of the sequence $x = GGCACTGAA$.

Solution: The matrix produced by the forward algorithm is given below, in log units (base 2). The first column is based on the start state, 0. The first entry of the second column is $log(f_H(1)) = log(e_H(G)) + log(1/2)$

	$-$	G	G	C	A	C	T	G	A	A
0	0									
H	∞	-2.51	-4.49	-6.39	-9.51	-10.49	-13.55	-14.53	-17.58	-19.78
L	∞	-3.32	-5.08	-6.97	-8.27	-10.89	-12.30	-14.92	-16.33	-18.37

The log probability is thus $\log(P(x)) = \log(2^{-19.78} + 2^{-18.37}) = -17.91$.

18.3 The backward algorithm and calculating P(x)

The backward algorithm is similar and if we run it to the end, we again calculate $P(x)$. This starts from the end of the sequence and works back to the beginning. Define

$$b_k(i) = P(x_{(i+1):L}|\pi_i = k)$$

the probability observing the last part of a sequence, $x_{i+1}, x_{i+2}, \ldots, x_L$ conditional on starting in state k at time i.

Once again, this is calculated using tabular computation:

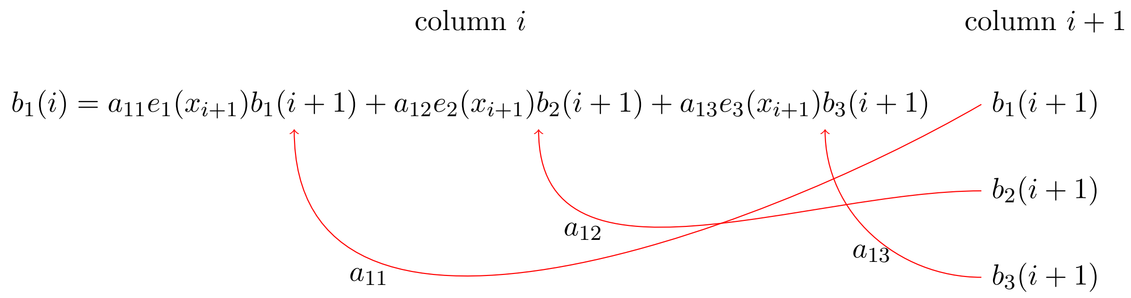
Initialisation $i = L$: $b_k(L) = a_{k0}$ for all k.

Recursion $i = L-1, \ldots, 1$: $b_k(i) = \sum_l a_{kl} e_l(x_{i+1}) b_l(i+1)$.

Termination: $P(x) = \sum_l a_{0l} e_l(x_1) b_l(1)$.

If we don't model the end of the sequence, $a_{k0} = 1$.

Here's a diagram showing how to get the ith column from the $(i+1)$th column in the backward algorithm. The example shown has three states 1, 2 and 3.


column i
column $i+1$
$b_1(i) = a_{11}e_1(x_{i+1})b_1(i+1) + a_{12}e_2(x_{i+1})b_2(i+1) + a_{13}e_3(x_{i+1})b_3(i+1)$
$b_1(i+1)$
$b_2(i+1)$
$b_3(i+1)$
a_{11}
a_{12}
a_{13}

The log version of the algorithm is (writing $B_k(i) = \log b_k(i)$):

Initialisation $i = L$**:** $B_k(L) = A_{k0}$ for all k (or $B_k(L) = 0$ if end not modelled)

Recursion $i = L-1, \ldots, 1$**:** $B_k(i) = \log\left[\sum_l \exp(A_{kl} + E_l(x_{i+1}) + B_l(i+1))\right]$.

Termination: $P(x) = \log\left[\sum_l \exp(A_{0l} + E_l(x_1) + B_l(1))\right]$.

18.4 The posterior probability of being in state k at time i $P(\pi_i = k|x)$

The final product of this backward algorithm is not usually what we are interested in (we use the forward algorithm to calculate that) but the combined forward and backward algorithms allow us to calculate the joint probability of the all observations and the prob that $\pi_i = k$

$$\begin{aligned} P(x, \pi_i = k) &= P(x_{1:i}, \pi_i = k)P(x_{i+1:L}|x_{1:i}, \pi_i = k) \text{ since } P(A,B) = P(A)P(B|A) \\ &= P(x_{1:i}, \pi_i = k)P(x_{i+1:L}|\pi_i = k) \text{ by Markov property} \\ &= f_k(i)b_k(i) \end{aligned}$$

Of more interest is the posterior probability $P(\pi_i = k|x)$ which we obtain directly by

$$P(\pi_i = k|x) = \frac{f_k(i)b_k(i)}{P(x)},$$

where the denominator is calculated either from the forward or backward algorithm.

18.5 What can we do with the posterior estimates?

We saw that the Viterbi path, π^*, is the most likely single path. But usually the most likely path is not very likely at all — there may be many other paths the are nearly as likely. We can use the posterior $P(\pi_i = k|x)$ to get some other likely paths.

The first is $\hat{\pi}$, the maximum posterior path, where

$$\hat{\pi}_i = \arg\max_k P(\pi_i = k|x).$$

Note that $\hat{\pi}$ is often not a legal path through the state space as it may include transitions that are not allowed.

The second is when we are interested in some function of the states, $g(k)$. In these cases, we calculate the the posterior expectation of g at a particular position,

$$G(i|x) = E_k[g(\pi_i|x)] = \sum_k P(\pi_i = k|x)g(k).$$

In particular, if g is an indicator function, that is, g takes the value 1 for some subset of states and 0 for all others, $E_k[g(\pi_i|x)]$ is just the posterior probability that π_i is in the specified subset.

18.6 Estimating the parameters of an HMM

So far we have assumed we know the structure of the HMM and the associated parameter values (the transition probabilities a_{kl} and emission probabilities $e_k(b)$). In general, we don't know either of these. What we usually do is decide on a model (based on our knowledge of the system) and then estimate the parameters of the model. Let $\theta = \{a_{kl}, e_k(b)\}$ be the set of all parameters of the model.

Then we are interested in finding the set of parameters that maximizes the (log) likelihood

$$l(x^1, \ldots, x^n|\theta) = \log P(x^1, \ldots, x^n|\theta) = \sum_{j=1}^{n} \log P(x^j|\theta).$$

The likelihood of the jth sequence, $P(x^j|\theta)$, is just what we have been referring to as $P(x^j)$ up to this point as we had always assumed the parameter values, θ, were known. Writing it as $P(x^j|\theta)$ simply emphasises the fact that we think of it now as a function of the unknown θ.

If we knew the state paths for a long sequence (or many short sequences), we could estimate the parameters simply by using the empirical proportions of transitions and emissions as our probabilities:

$$\hat{a}_{kl} = \frac{A_{kl}}{\sum_i A_{ki}} \text{ and } \hat{e}_k(b)\frac{E_k(b)}{\sum_j E_k(j)}$$

where A and E are empirical counts. Note that, as some transitions or emissions probably wouldn't occur in smaller datasets, it is advisable add a small number of 'pseudo-counts' to the empirical counts so that none are zero). But assuming we know the state paths is unrealistic, so we proceed assuming we have only observed sequences $x^1, x^2, \ldots, x^n$.

18.7 Baum-Welch algorithm for estimating parameters of HMM

The Baum-Welch algorithm is an iterative algorithm that attempts to maximize the (log) likelihood of an HMM. Unlike earlier algorithms we have seen, it is not exact, so the estimate it finds is not guaranteed to be the best. It may also get stuck in local maxima, so different starting points are necessary.

The idea of the algorithm is to pick a starting value for $\theta = (a, e)$. Probable paths for this value of θ are found. From these probable paths, a new value for θ is found by calculating A and E. This process repeats until the likelihood of θ converges on some value (that is, no change or a very small change is seen in $l(x^1, \dots, x^n|\theta)$ from one step to the next).

In one version of the algorithm, the probable paths used are the Viterbi paths for each sequence and the values A and E are calculated from these paths. This seems reasonable and can produce satisfactory results but it does not converge to the maximum likelihood estimate.

It turns out that we can avoid actually imputing a probable path by directly calculating the probability that the transition from k to l occurs at position i in x:

$$P(\pi_i = k, \pi_{i+1} = l|x, \theta) = \frac{f_k(i) a_{kl} e_l(x_{i+1}) b_l(i+1)}{P(x)}.$$

Thus, to get a the expected value of A_{kl}, we simply sum over all possible values of i. A similar argument can be made for E. The expected values for A_{kl} and E_{kl} are then:

$$A_{kl} = \sum_j \frac{1}{P(x^j)} \sum_i f_k^j(i) a_{kl} e_l(x_{i+1}^j) b_l^j(i+1) \quad (7)$$

$$E_k(b) = \sum_j \frac{1}{P(x^j)} \sum_{i:x_i^j = b} f_k^j(i) b_k^j(i) \quad (8)$$

The Baum-Welch algorithm proceeds as follows:

Initialise: Set starting values for the parameters. Set log-likelihood to $-\infty$.

Iterate:

1. Set A and E to their pseudo count values. For each training sequence x^j:
   (a) Calculate $f_k(i)$ for x^j from forward algorithm
   (b) Calculate $b_k(i)$ for x^j from backward algorithm
   (c) Calculate A^j and E^j and add to A and E using Equations 7 and 8 above.
2. Set new values for a and e, based on A and E
3. Calculate log-likelihood of model
4. If change in log likelihood is small, stop, else, continue.

See lecture slides for a detailed example of applying the Baum-Welch algorithm.

18.7.1 Comments on the Baum-Welch algorithm

The Baum-Welch algorithm is guaranteed to converge to the local maximum — exactly which local maximum it converges to depends on the initial state. Any local maximum is not necessarily the global maximum so the algorithm should be run from multiple different start states to check that a global maximum has been found.

Also remember that convergence is only guaranteed in the limit of an infinite number of iterations so the exact local maximum is never achieved.

The algorithm is a type of Expectation-Maximisation (EM) algorithm that is widely used for maximum likelihood estimation.

18.8 Sampling state paths

The probabilities $f_k(i)$ we calculate in the forward algorithm can be used to sample possible state paths in proportion to their probability. Recall that the Viterbi algorithm provides a method of finding the most probable state path by tracing back though the a matrix, taking the direction that led us to the highest score at each point. We adopt the traceback idea but apply it to the matrix $f_k(i)$ and at each step of the traceback, we choose the state in proportion to the amount it contributed to the current probability.

Assuming an end state is not modelled, the probability of a sequence x is $P(x) = \sum_k f_k(L)$. So the probability that the last state is k is given by

$$P(\pi_L = k|x) = \frac{f_k(L)}{\sum_i f_i(L)}.$$

Now, suppose we are in state l at position $i+1$. We know from the forward algorithm that

$$f_l(i+1) = e_l(x_{i+1}) \sum_k f_k(i) a_{kl}.$$

Thus we move to state k in the ith position with probability

$$\frac{f_k(i) a_{kl}}{\sum_j f_j(i) a_{jl}} = \frac{e_l(x_{i+1}) f_k(i) a_{kl}}{f_l(i+1)}.$$

Depending on what you have stored in your algorithm, it may be easier to work with either the left or right hand side of this equation.

Example: Looking again at the CpG island example, sample state paths according to their posterior probabilities for the given sequence $x = TACA$.

Solution: First, get the forward matrix, f. To make it simple, don't use the log transform:

	-	T	A	C	A
0	1				
H	0	0.075	0.014625	0.007914375	0.0009567281
L	0	0.150	0.038250	0.006052500	0.0022766062

Now, $P(x) = \sum_k f_k(L) a_{k0} = 0.003233334$ where $a_{k0} = 1$. So simulate the 4th element of the state path by drawing from $\{H, L\}$ with probabilities $\{\frac{f_H(4)}{P(x)} = 0.2958952, \frac{f_L(4)}{P(x)} =$

0.7041048}, respectively. Suppose we sampled H. Then the 3rd element of the state path is a draw from $\{H, L\}$ with respective probabilities

$$\left\{\frac{f_H(3)a_{HH}}{f_H(3)a_{HH}+f_L(3)a_{LH}} = 0.6204251, \frac{f_L(3)a_{LH}}{f_H(3)a_{HH}+f_L(3)a_{LH}} = 0.3795749\right\}$$

Suppose we sampled H again. We now repeat the process, sampling from

$$\left\{\frac{f_H(2)a_{HH}}{f_H(2)a_{HH}+f_L(2)a_{LH}}, \frac{f_L(2)a_{LH}}{f_H(2)a_{HH}+f_L(2)a_{LH}}\right\}$$

and so on. We'll end up with a state path, for example, $LLHH$. □

18.9 HMM model structure

Defining the correct structure, or 'topology', of an HMM is crucial to good estimation but there are no solid rules for doing so. Usually data is limited so we can't over-parametrise otherwise our estimation algorithms with never find decent values. So we can't simply allow all possible transitions and let the computer estimate the correct model. We must decide ourselves, as much as possible, which transition we allow (so that $a_{kl} > 0$) and which we disallow (by setting $a_{kl} = 0$).

18.9.1 Duration modeling

If we want to accurately model the length of a sequence along with the contents, we must model an end state as well as the start state. The basic end state, which is connected to every other state, and has transition probability $q = 1 - p$ produces a sequence of length l with probability

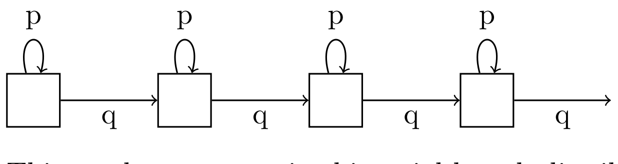
$$P(L = l) = qp^{l-1}.$$

This is a geometric distribution and is the discrete analogue of the exponential distribution. In general, it is not a very good model for lengths and is used largely for the convenience of its mathematical form.

Also note that the length of time an HMM spends in any one state where the probability of leaving that state is q is geometric.

An easy way to get a more flexible and, perhaps, more realistic distribution of lengths is to have, say, n copies of the HMM linked together an it stays in each one for a geometrically distributed number of steps.

An example with 4 states linked together (the states here could be HMMs themselves).



This produces a negative binomial length distribution, so that

$$P(L = l) = \binom{l-1}{n-1} p^{l-n} q^n.$$

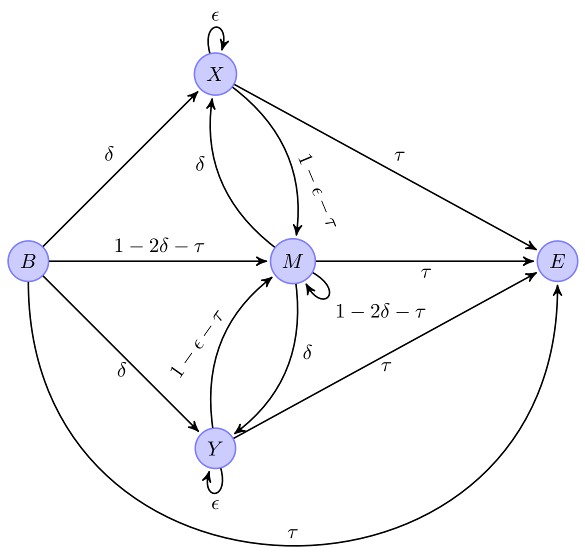


Figure 13: A pair HMM model for global alignment. Emission probabilities for states M, X, Y are $p_{x_i y_j}$, q_{x_i} and q_{y_j}, respectively. Compare it to the simpler FSA in Figure 9.

19 Applications of HMMs in bioinformatics

19.1 Pairwise alignment with HMMs

We saw that we could tackle the pairwise alignment problem with finite state automata. We now tackle the problem using an HMM, which is sometimes called a stochastic FSA.

We define a *pair HMM* as emitting a pair of sequences (x, y) as opposed to the standard HMMs we have considered so far that emit a single sequence.

The basic HMM which produces a global alignment has three states, X, Y and M. M emits a match, X emits a residue from sequence x and a gap in sequence y (an insertion in x relative to y), while Y emits a residue from y and a gap in x. Emission probabilities for states M, X, Y are $p_{x_i y_j}$, q_{x_i} and q_{y_j}, respectively. We also include begin and end states, B and E. Non-zero transition probabilities are $a_{MX} = a_{MY} = \delta$, $a_{XX} = a_{YY} = \epsilon$, $a_{BX} = a_{BY} = \delta$, $a_{kE} = \tau$ for any k, and $a_{kM} \neq 0$ for $k \neq E$ and can be calculated using the fact that $\sum_k a_{ik} = 1$.

All the algorithms we saw for standard HMMs will work for these pair HMMs but we need a little bit of accounting for the 2 sequences — instead of $v_k(i)$ or $f_k(i)$, we need to work with $v_k(i, j)$ or $f_k(i, j)$ etc where k is one of the 3 states M, X or Y. In each case, we keep 3 score matrices instead of 1 to keep track of which state we are in at every point in the alignment.

Durbin et al show how the parameters in these pair HMMs work within the Viterbi algorithm to produce exact analogues of the dynamic programming algorithms we saw earlier.

For example, to the standard quantities we use in the Needleman-Wunsch algorithm from the Viterbi HMM formulation of global alignment, set

$$\begin{aligned} s(a,b) &= \log \frac{p_{ab}}{q_a q_b} + \log 1 - 2\delta - \tau(1-\eta)^2 \\ d &= -\log \frac{\delta(1-\epsilon-\tau)}{(1-\eta)(1-2\delta-\tau)} \\ e &= -\log \frac{\epsilon}{1-\eta}. \end{aligned}$$

These pair HMMs give us more than just another way of viewing the basic alignment algorithms. Since they are couched in the language of probability, we can answer questions about alignments with more depth and nuance than we can with the standard deterministic tools.

19.1.1 Probability that two sequences are related

For example, given a pair sequences, x and y, we can ask what probability that two sequences are related without relying on any particular alignment. This is given by the quantity

$$P(x, y) = \sum_{\pi} P(x, y, \pi)$$

where the sum is over all possible alignments, π. As in the standard HMM case, we calculate this quantity via the forward algorithm or the backward algorithm.

19.1.2 Sampling alignments

Further, instead of relying on a single alignment, we could sample possible alignments in proportion to their probability using the techniques described in Section 18.8.

That is, we could calculate the forward $f_M(i, j)$, $f_X(i, j)$ and $f_Y(i, j)$ from the forward algorithm and then traceback to find a state path which is an alignment. The traceback from the current state chooses the next state (M, X, or Y) at each step according to it's contribution to the probability of the current state. This is exactly the technique described in the Section 18.8.

19.1.3 Probability that x_i and y_j are aligned

Consider two residues, x_i and y_j, in our given sequences x and y. What is the probability that they are actually aligned to each other? Write $\langle x_i, y_j \rangle$ to mean x_i and y_j are aligned, so the probability of interest is $\Pr(\langle x_i, y_j \rangle | x, y)$. We could estimate this probability by sampling many alignments using the technique described above and counting the proportion of times that $\langle x_i, y_j \rangle$.

But it turns out that we can calculate this number exactly using the general technique of finding the posterior probability of being in a state at some time described in Section 18.4. x_i is aligned to y_j exactly when the HMM is in state M at (i, j). Thus

$$\Pr(\langle x_i, y_j \rangle | x, y) = P(\pi(i, j) = M | x, y) = \frac{P(\pi(i, j) = M, x, y)}{P(x, y)} = \frac{f_M(i, j) b_M(i, j)}{P(x, y)}.$$

The last equality is exactly the same as the one derived in Section 18.4 and, as usual, the quantity $P(x, y)$ can be calculated using either the forward or backward algorithm.

19.2 Profile HMMs

The canonical problem in genetics is to find a group of sequences that are homologous. In particular, we are interested in finding homologous genes that share a similar function. We say such sequences or genes belong to the same family in the sense that they share a common ancestor and have maintained the same (or similar) functionality. They may be the same sequence in different species or in the same species but in different parts of the genome (having arrived there through duplication). Sequences in the same family will often have features in common, particularly where they share the same function and, therefore, the same basic secondary structure.

If we can characterise these families accurately, by finding features that almost certainly share in common and identifying regions where more variation is seen, we will be able to better align sequences known to belong to the family and more easily identify other members of the family. We achieve this characterisation by modelling the family using an HMM, known as a *profile HMM*.

We'll start by assuming that we are given a family of homologous sequences that are already aligned into a multiple sequence alignment (MSA). See a very small example in Figure 14.

```
VGA--HAGEY
V----NVDEV
VEA--DVAGH
VKG------D
VYS--TYETS
FNA--NIPKH
IAGADNGAGV
```

Figure 14: Ten columns from a given multiple alignment of seven globin sequences.

From a given alignment, we wish to characterise a typical sequence in the alignment at each position. Once we have made this characterisation, we could use it to search for other sequences (or parts of sequences) that fit the profile and so are candidates to be members of this family.

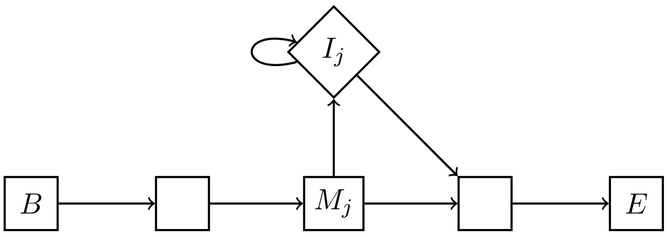
We model the alignment as an HMM, where each position in the alignment is a state with a distinct probability of emitting the various residues.

We'll start by supposing the alignment that is largely free of gaps (by ignoring, say, columns in the alignment that are more than 50% gaps). With each position in the alignment, associate a state in the HMM. Call this state a match state and label the ith match state M_i. The (ungapped) alignment is then modelled as a HMM with only the trivial transitions, M_i to M_{i+1} (with additional begin and end states). Let emission probabilities from the i match state be e_{M_i}. A model for a MSA of length 3 looks like:

$$\begin{tikzcd}
\boxed{B} \arrow[r] & \boxed{M_1} \arrow[r] & \boxed{M_2} \arrow[r] & \boxed{M_3} \arrow[r] & \boxed{E}
\end{tikzcd}$$

Now let's allow gaps in the alignment. How do we handle them? To handle an insertions (with respect to the alignment — parts of a sequence x that are not matched by anything in the model) we introduce a new set of states I_i which matches the residues in x after i to a gap. These states have emission probabilities $e_{I_i}(a)$ which we set to the background rate: $e_{I_i}(a) = q_a$.

There is a transition from M_i to I_i, a loop at I_i, and a transition from I_i to M_{i+1}.

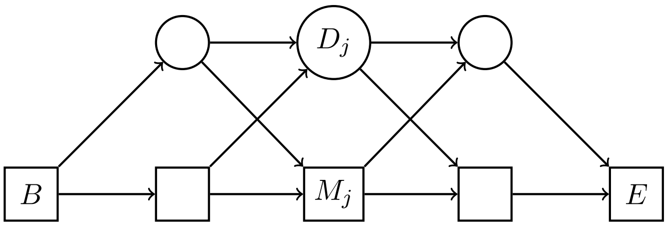


A gap of length k therefore has log-odds score

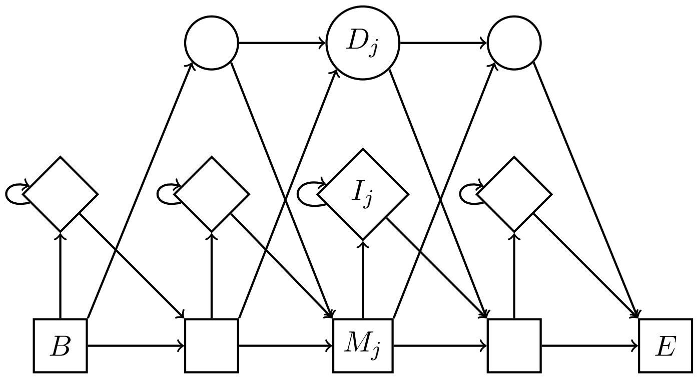
$$\log a_{M_i I_i} + \log a_{I_i M_{i+1}} + (k-1) \log a_{I_i I_i},$$

the same as an affine gap penalty.

A deletion relative to the model corresponds to skipping ahead in the model. To allow transitions from every match state to every other state ahead of it in the model would introduce too many transitions (how many would we need?) so we introduce instead a silent delete states D_i next to every match state. Silent states emit no residues. We allow transitions $a_{M_{i-1} D_i}, a_{D_i M_{i+1}}$ and $a_{D_i D_{i+1}}$.



A full model incorporates both insert and delete states and is drawn below. Notice that we have not drawn transitions between I and D states though it simplifies computation to allow them (we'd allow $D_j \to I_j$ and $I_j \to D_{j+1}$).



This profile HMM model is equivalent to a pair HMM model for pairwise alignment where the given MSA is summarised into a single sequence y . So when we seek to fit a candidate sequence x to our profile HMM, it is as though we are aligning x and y with a pair HMM (with appropriately chosen emission probabilities).

19.2.1 Estimating the parameters of a profile HMM

We can choose the parameters of the profile HMM using the empirical counts (A_{kl} and $E_k(b)$) with pseudo-counts added. Remember that the number of possible transitions is limited: in our full drawn model we only allow non-zero transitions for $a_{M_iM_{i+1}}$, $a_{M_iD_{i+1}}$, $a_{M_iI_i}$, $a_{I_iI_i}$, $a_{I_iM_{i+1}}$, $a_{D_iD_{i+1}}$ and $a_{D_iM_{i+1}}$. We have emissions for all possible residues at each site so we add pseudo-counts to ensure $e_{M_i}(a) > 0$ for all a. Then we use the rule we saw earlier to estimate transition and emission probabilities:

$$a_{kl} = \frac{A_{kl}}{\sum_j A_{kj}} \text{ and } e_k(b) = \frac{E_k(b)}{\sum_j E_k(j)}.$$

A simple way of assigning pseudo-counts is to add 1 to all scores (including zero scores). There is a lengthy discussion of which pseudo-count values to choose in the Durbin et al book and many more sensible schemes are proposed – none are particularly complicated but we don't have time to cover them all here.

We assume emissions from insert states are at the background rate (that is, that rate the residue occurs at at any position).

In the example in Figure 14, the first column has all seven sequences in the match state with 5 Vs, and 1 each of F and I. The 17 other possible residues are not observed, so have frequency 0. Adding a pseudo-count of 1 to each observed frequency gives us emission probabilities $e_{M_1}(V) = 6/27$, $e_{M_1}(F) = e_{M_1}(I) = 2/27$ and the other 17 residues have $e_{M_1}(b) = 1/27$. 6 of the 7 transitions to the next column are to the match state again while one is to a delete state. Again, adding pseudo-countsof one gives $a_{M_1M_2} = 7/10$, $a_{M_1D_2} = 2/10$ and $a_{M_1I_1} = 1/10$. Transitions from the insert and delete states are just based on the pseudo-counts here.

19.2.2 Finding matches

Once the model has been set and the parameters estimated, we can set about seeing whether other sequences match the family. Call M the model and x a sequence we wish to test against the model. To do so, we align proposed sequences to the family using the now familiar tools of the Viterbi algorithm, giving the Viterbi path π^* (and the associated score, $P(x|\pi^*, M)$) or, better, the full probability of a sequence summed over all alignments, $P(x|M)$.

These scores can be used to compare the hypotheses that x belongs to family M or x is no more like M than we would expect at random. Call R the random model and let $P(x|R) = \prod_{i=1}^{L} q_{x_i}$ be the likelihood of x under the random model where q_a is simply the background rate of occurrence of residue a. If the log ratio $\log(P(x|M)/P(x|R)) > 0$, x is more likely associated with the modelled family than not. To find more certain matches, we may chose a threshold that this ratio must exceed before we call it a member of the family.

Note that if we wish to fit new found sequence x to the family, we can simply use the Viterbi alignment to align it to the family and update parameters of the model.

19.2.3 Alignment with a known profile HMM

The simplest case is when we have a known aligned family to which we have already fitted a profile HMM and we wish to add a number of sequences to the profile. In this case, we use the Viterbi algorithm to find the most probable alignment for the new sequences.

The Viterbi path will consist of matches, insert and delete states. At the delete states, we add a gap character, –, to the sequence we are aligning. At an insert state, the unaligned residue of the sequence we are aligning is emitted, forcing a gap like character in the already aligned profile. In the profile, we placeholder character, ., at these positions. Note that if there are multiple insertions in different sequences at a position, there are many ways to align them against each other. Since we believe insertions are not shared between all members of the family, we can set an arbitrary rule for aligning them, such as using simple left-justification.

19.2.4 Alignment from unaligned sequences with HMMs

If we do not have a pre-existing aligned family and/or profile HMM, we need to first specify an HMM and then use the Baum-Welch algorithm to estimate the parameters. After we have done that, we are in the same position as above and can construct the MSA from the fitted profile HMM.

A rule of thumb for specifying the HMM in the absence of prior knowledge is to allow M match states where M is the average length of the training sequences. In general, it is difficult to fit an HMM of this size. A number of heuristics have been developed to avoid local maxima.

ClustalΩ implements this method and is quick enough to align thousands of sequences in reasonable time. Currently, ClustalΩ can only be used for protein sequences. See Fabian Sievers et al, 2011, Molecular Systems Biology 7, `http://www.nature.com/msb/journal/v7/n1/full/msb201175.html` for a full description.

19.3 Gene finding

A DNA sequence can roughly be divided into two types of region: genes and non-genes or inter-genic regions. Genes are regions that code for proteins which actually perform biological functions in an organism so these regions are of primary interest. The role played by intergenic regions is not yet clear and it is often referred to as 'junk DNA'.

We are interested, then, to find a method of finding regions of a given sequence that are genes. To do so, we need to look at how a gene is structured along a sequence. Recalling that a sequence has a direction with one end being 5' end, the other being the 3' end. Moving forward in the sequence is going from the 5' end towards the 3' end.

Our model of a gene has the following elements occurring in the order listed here: an inter-genic region, a promoter region, a 5' un-transcribed region, a series of *exons* and

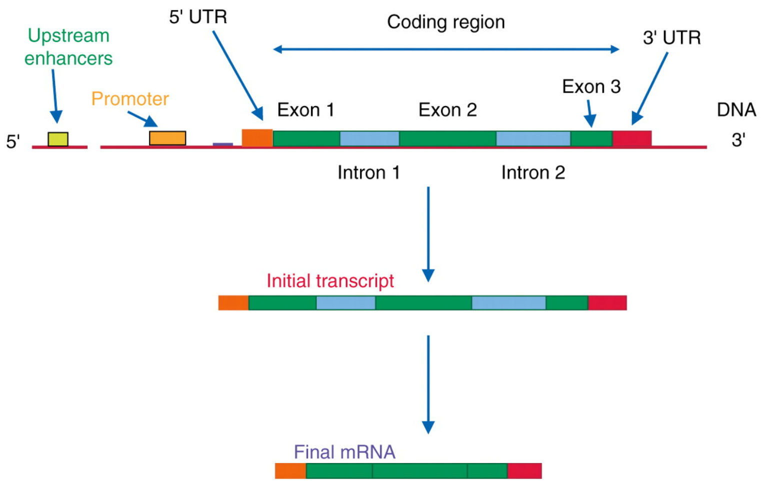


Figure 15: Gene structure and transcription. The DNA of the coding region is composed of exons (coding DNA) interspersed with introns (non-coding DNA) and is flanked by untranslated regions (UTRs). Upstream of the coding regions within the gene are DNA sequences that control (promoter) and regulate (enhancers) gene expression. During transcription, the initial nuclear transcript includes RNA sequence complementary to the entire coding region and the UTRs. In a subsequent step, the introns are spliced out to form mRNA which translocates to the cytoplasm where it is translated into protein. Source: `http://dx.doi.org/10.1093/bja/aep130` by UoA subscription

introns, a 3' un-transcribed region, a poly-A region and then back to another inter-genic region.

These regions are described in more detail below.

Inter-genic region: A non-coding region between genes.

Promoter: a region of DNA that facilitates the transcription of a particular gene. Promoters are located near the genes they regulate, on the same strand and typically upstream (towards the 5' region of the sense strand). For the transcription to take place, the enzyme that synthesizes RNA, known as RNA polymerase, must attach to the DNA near a gene. Promoters contain specific DNA sequences and response elements that provide a secure initial binding site for RNA polymerase and for proteins called transcription factors that recruit RNA polymerase. These transcription factors have specific activator or repressor sequences of corresponding nucleotides that attach to specific promoters and regulate gene expressions.

As promoters are typically immediately adjacent to the gene in question, positions in the promoter are designated relative to the transcriptional start site, where transcription of RNA begins for a particular gene (i.e., positions upstream are negative numbers counting back from -1, for example -100 is a position 100 base pairs upstream).

In eukaryotes, the process is complex and promoters may occur hundreds of base-pairs upstream.

In prokaryotes, the promoter consists of two short sequences at -10 (that is 10 bases upstream from the UTR and is called the Pribnow box which typically looks like TATAAT) and at -35 (the -35 element, usually TTGACAT). The promoter regions are not transcribed to RNA.

Untranslated regions (UTR 5' or 3') : These are regions immediately flanking the translated region. They are transcribed into RNA but not translated into proteins.

Exons and introns: An exon is transcribed into RNA and is further translated into a protein. An intron is transcribed into a form of RNA and then spliced out of the RNA sequence that finally gets translated in a protein. In any gene, there could be one or many exons and zero or many introns. Exons and introns alternate along the sequence.

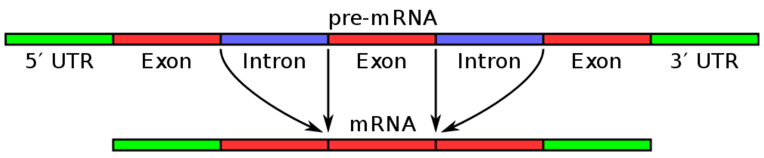


Figure 16: A figure showing how the transcribed precursor to messenger RNA includes the UTRs, exons and introns. The introns are spliced out to form the messenger RNA. The Exons in the mRNA are translated into proteins. Souce: `http://en.wikipedia.org/wiki/File:Pre-mRNA_to_mRNA.svg`

Poly(A) signal: After the 3' UTR on the RNA, a number of adenines (As) is added — this is called the poly(A) tail. The poly(A) signal, or polyadenylation signal, is thus a stretch of DNA that signals to the RNA transcription mechanism to begin the addition of the poly(A) tail.

A method first described in Burge and Karlin 1997 (see `http://www.ncbi.nlm.nih.gov/pubmed/9149143`) describes a generalized HMM incorporating all these regions. See Figure 17 for a sketch of the structure of the HMM. The states of the HMM are N (corresponding to an inter-genic region), P (promotor), F (5' UTR), E (exons), I (introns), T (3' UTR) and A (poly(A) signal). The multiple states for introns, I_0, I_1 and I_2 and exons E_0, E_1 and E_2 indicate the relation of the reading frame of the exon to the reading frame of the initial exon (E_{init}). Recall that three bases of DNA code for a single amino acid, with each group of 3 bases call a codon. If an exon or intron does not have length that is a multiple of 3, then the start of the next exon or intron may be out of phase with it. A subscript of 0, 1 or 2 represents in phase or lagging 1 or 2 steps out of phase, respectively.

The GHMM produces a set of states $q = q_1 \dots q_n$ with an associated set of lengths (durations) $d = d_1 \dots d_n$ and for each state q_i, it produces a sequence of length d_i according to a probability model associated with the state q_i. Algorithms to analyse sequences according to this model are implemented in Genscan and GlimmerHMM.

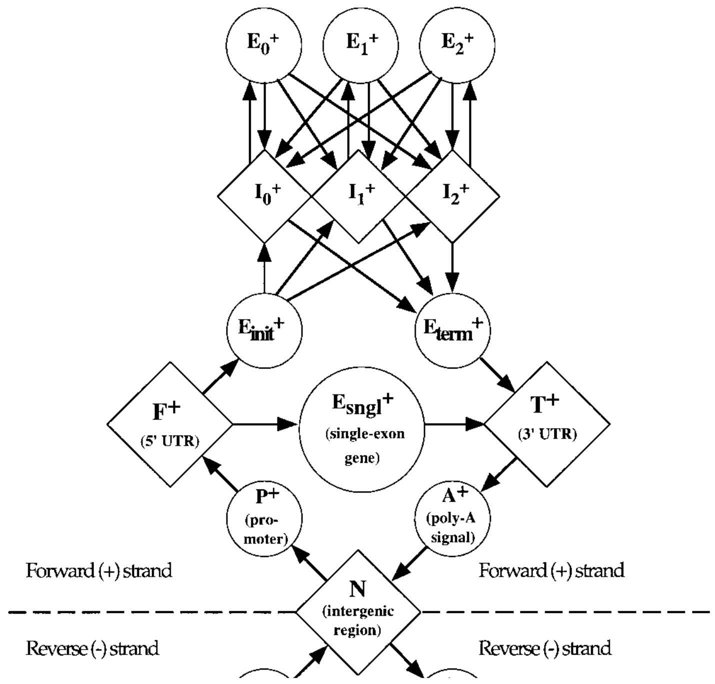


Figure 17: The structure of the (generalised) HMM used for gene finding from Burge et al 1997. See text for details. The full HMM includes a mirror image corresponding to the reverse strand which has been deleted here.

20 Reconstructing trees

The fastest ways of constructing trees rely on defining a distance between sequences. We have already one method that does this: UPGMA in Section 17.3. We looked at UPGMA in the context of multiple sequence alignment where a sensible choice of distance between seqeces to use was $D(x, y) = -\log S_{eff}(x, y)$. We'll briefly look at other, more widely used distance measures.

20.1 Defining distances between sequences

There are numerous ways of defining distances between sequences. The simplest, for an aligned pair of sequences x and y of length L is to count the number of positions where they differ, D_{xy} say, and define the distance to be

$$f_{xy} = D_{xy}/L,$$

which is simply the fraction of sites at which they differ. This method works well for related sequences where f is expected to be small, but doesn't grow as much as we would like as sequences become less and less related since even unrelated sequences share many bases in common due to chance.

The Jukes-Cantor distance is based on the simplest model of sequence evolution where mutations between all four bases are equally likely. The distance includes a correction for the fact that unrelated sequences will agree simply due to chance. The distance is defined by

$$d_{xy} = -\frac{3}{4}\log(1 - \frac{4f_{xy}}{3}).$$

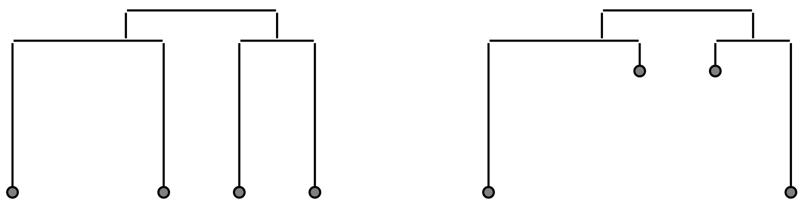
Since the background level of dissimilarity (given by f_{xy}) for unrelated sequences is $\frac{3}{4}$, $(1 - \frac{4f_{xy}}{3})$ tends to zero as sequences become more unrelated so d_{xy} tends to infinity for unrelated sequences.

20.2 Ultrametric distances

UPGMA produces the correct tree (i.e., produces the tree along which the sequences actually evolved) if the sequences evolved according to a *molecular clock* in which sequences evolved at a constant rate over the whole tree. In that case, the number of mutations is proportional to the temporal distance of a node to the ancestor.

In these cases, the distances are said to be *ultrametric* and UPGMA will reconstruct the correct tree. The ultrametric condition is that d_{ij} is ultrametric when, for and points i, j, k, the distances d_{ij}, d_{jk}, d_{ik} are either all equal or two are equal and the remaining one is smaller.

More simply, in an ultrametric tree, the distance from the root to the leaves is the same for every leaf. So if all leaf nodes are sampled at the same time and the ultrametric property holds, the tree displaying the distances will have all the leaf nodes at the same level.



The tree on the left is ultrametric, the tree on the right is not.

In most cases, the ultrametric assumption is not a good one, as different regions of sequences vary at different rates and different lineages of the tree may have different rates of mutation. Thus, UPGMA will not reconstruct the correct tree in most cases.

20.3 Additive distances

A less stringent condition is that distances are *additive*. A tree is said to have additive edge lengths if the distance between two leaves is the sum of the edge lengths connecting them. You can show that ultrametric distances are additive but the reverse does not hold. In an additive tree, the *four point condition* is satisfied, in which any four leaves can be relabelled so that $d(x, y) + d(u, v) \leq d(x, u) + d(y, v) = d(x, v) + d(y, z)$.

A set of additive distance can be thought of as tree-like — there is a tree that correctly displays those distance as branch lengths.

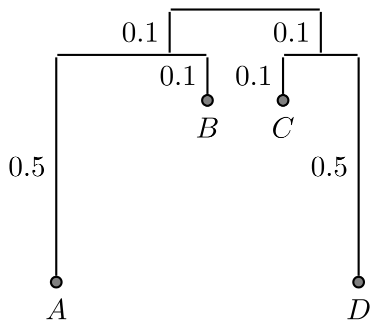
So the question is, given a set of additive distances, can we reconstruct the correct tree?

20.4 Neighbour joining

The answer turns out to be yes, and the algorithm that lets us achieve this is known as *neighbour joining* (NJ). NJ is similar to UPGMA but instead of simply using a pairwise evolutionary distance matrix, NJ takes that matrix as a starting point and then builds a rate-corrected distance matrix before proceeding to join nearest neighbours.

First, note that to find the nearest neighbour on a tree, it is not sufficient to simply calculate the smallest distance.

Example: Consider the tree



from which we derive the pairwise distance matrix

$$d = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 0 & 0.6 & 0.8 & 1.2 \\ & 0 & 0.4 & 0.8 \\ & & 0 & 0.6 \\ & & & 0 \end{pmatrix} \end{array}.$$

If we try to reconstruct the tree using UPGMA, the first step is to choose the pair with the smallest distance and join them. That has us choosing B and C first as sharing a common ancestor before anything else. This immediately leads to the the wrong tree topology. □

To find the nearest neighbour instead of just the node at the smallest distance, we need to subtract the average distance to all other leaves. Let $D_{ij} = d_{ij} - (r_i + r_j)$ where

$$r_i = \frac{1}{|L| - 2} \sum_{k \in L} d_{ik}$$

and L is the number of leaves (sequences). Note that the denominator in calculating r is deliberately one less than the number of items summed. It can be shown (with a bit of work, omitted here) that the pair of leaves i, j for which D_{ij} is minimal are neighbouring leaves.

This leads us to the NJ algorithm, which progressive builds up a tree T by keeping a list of active nodes L and finding the closest amongst them.

Neighour joining algorithm

1. Let T be the set of all leaf nodes and set $L = T$.
2. Iterate until $|L| = 2$:
    (a) Calculate (or update) D from the distance matrix d.
    (b) Pick i, j for which D_{ij} is minimal.
    (c) Define k so that $d_{km} = \frac{1}{2}(d_{im} + d_{jm} - d_{ij})$ for all $m \in L$.
    (d) Add k to T with edges joining to i and j with lengths $d_{ik} = \frac{1}{2}(d_{ij} + r_i - r_j)$ and $d_{jk} = d_{ij} - d_{ik}$.
    (e) Set $L = L - \{i, j\} + k$.
3. $|L| = 2$, so add remaining edge connect i, j with length d_{ij}.

To see this works, consider the reverse process where we strip away leaves from an additive tree by removing neighbouring pairs. Find leaves i, j with parent k. Remove i, j and add k to the list of leaves, defining $d_{km} = \frac{1}{2}(d_{im} + d_{jm} - d_{ij})$ where m is some other leaf node.

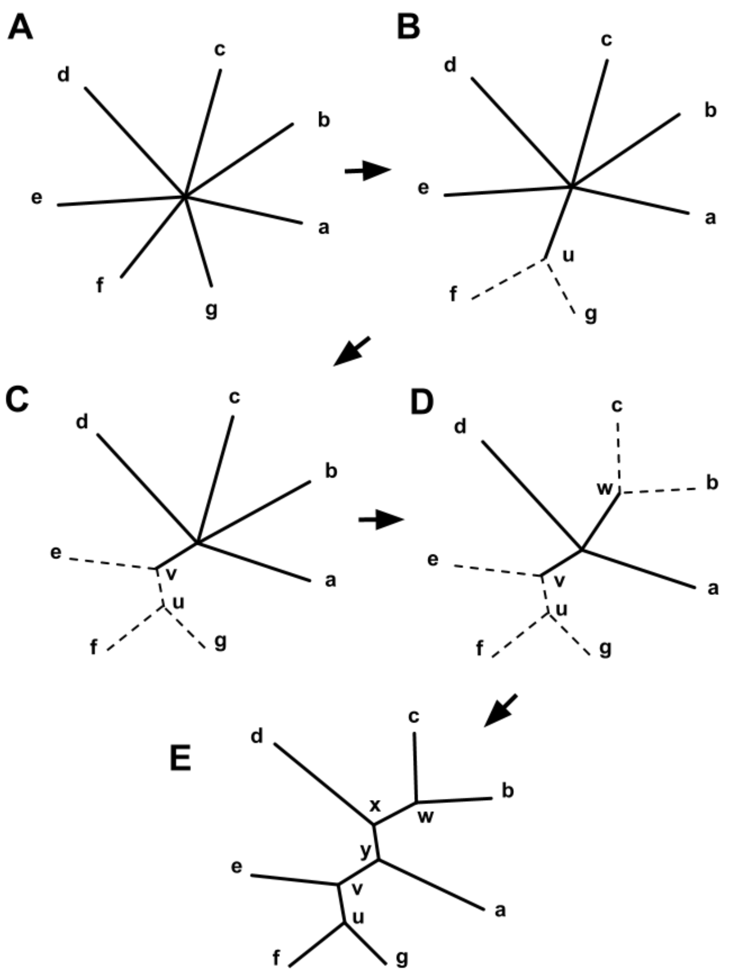


Figure 18: Example from wikipedia `http://en.wikipedia.org/wiki/Neighbor_joining`: Starting with a star tree in which all leaf nodes are active (A). The matrix D is calculated and used to choose a pair of nodes for joining, in this case f and g. These are joined to a newly created node, u, as shown in (B). The part of the tree shown as dotted lines is now fixed and will not be changed in subsequent joining steps. The distances from node u to the nodes a-e are computed from the formula given in the text. This process is then repeated, using a matrix of just the distances between the nodes, a,b,c,d,e, and u, and a new D matrix derived from it. In this case u and e are joined to the newly created v, as shown in (C). Two more iterations lead first to (D), and then to (E), at which point the algorithm is done, as the tree is fully resolved.

Example: Perform neighbour joining on the distance matrix from the previous example

$$d = \begin{array}{c|cccc} & A & B & C & D \\ A & 0 & 0.6 & 0.8 & 1.2 \\ B & & 0 & 0.4 & 0.8 \\ C & & & 0 & 0.6 \\ D & & & & 0 \end{array}.$$

Solution: We first need to calculate D for which we will need r. Here $L = 4$, so

$$r_A = \frac{1}{2}(d_{AB} + d_{AD} + d_{AD}) = \frac{1}{2}(0.6 + 0.8 + 1.2) = 2.6/2 = 1.3.$$

Get other elements of r similarly so $r = (1.3, 0.9, 0.9, 1.3)$.

From r and d we can thus calculate

$$D = \begin{array}{c|cccc} & A & B & C & D \\ A & - & -1.6 & -1.4 & -1.4 \\ B & - & - & -1.4 & -1.4 \\ C & - & - & - & -1.6 \\ D & - & - & - & - \end{array}.$$

D is symmetric and the diagonal is irrelevant so only need calculate either the elements of the upper or lower traingle.

The minimum value of the rate-adjusted matrix is found at AB and CD. Choose AB to merge into new node E. The length of the edge from A to E is $d_{AE} = \frac{1}{2}(d_{AB} + r_A - r_B) = \frac{1}{2}(0.6 + 1.3 - 0.9) = 0.5$ while the distance from B to E is found by $d_{AE} = d_{AB} - d_{AE} = 0.6 - 0.5 = 0.1$. Check these branch lengths against the values in the original tree: they match.

A and B can now be removed from the leaf set and replaced with E and a new rate adjusted matrix D derived. Completing a further iteration and the final step reconstructs the original tree. □

20.4.1 Unrooted vs rooted trees

Notice that the neighbour-joining algorithm produces a tree with no root. That is, we known branch lengths (in terms of distance between sequences — roughly, the number of changes along a branch) but we don't know the actual times of the nodes, so we don't know the position of the root. A tree with no root is an *unrooted tree* and a tree with a known root position is called a *rooted tree.*

In some cases we can determine the position of the root by including a known *out-group* in the analysis. For example, if we have samples from 20 hominids, we could include a chimp as an out-group since we know that the hominids all share a recent common ancestor before the most recent common ancestor of hominids and chimps. In this example, we would place the root on the branch separating the chimp from the hominids.

The number of trees, rooted or unrooted is huge. If we have n taxa, there are

$$\frac{(2n-5)!}{2^{n-3}(n-3)!}$$

unrooted trees and

$$\frac{(2n-3)!}{2^{n-2}(n-2)!}$$

rooted trees. So when $n = 5$, we have 15 unrooted and 105 rooted trees, but for $n = 10$ there are about 2 million unrooted and 3.5 million rooted trees.

20.4.2 Complexity of neighbour jointing and UPGMA

UPGMA has time and space complexity of $O(n^2)$ while neighbour-joining has the same space complexity but time complexity of $O(n^3)$.

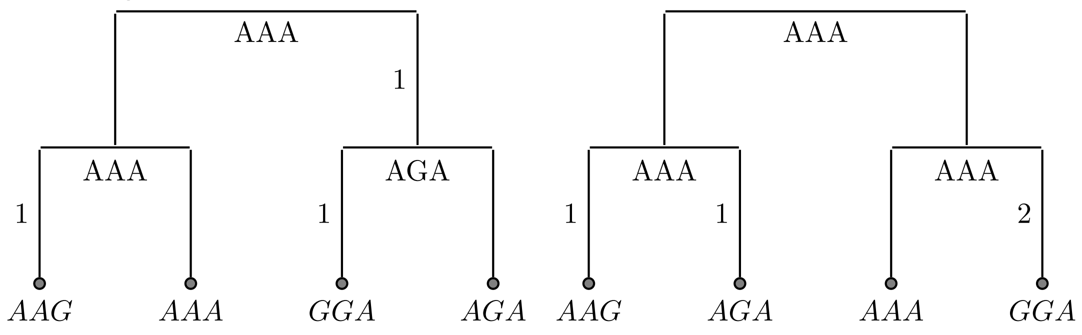
However, these are worst case complexities, and there are various heuristics that result in average time performance for neighbour-joining appears somewhat better than $O(n^3)$.

20.5 Parsimony

Parsimony is form of Occam's razor. It postulates that the best tree is the one that requires the fewest changes along it to explain all sequences. This best tree is called the most parsimonious tree or the *(maximum) parsimony tree.*

The main algorithm that we discuss here is not actually a method for constructing the maximum parsimony tree but rather provides a way of calculating the cost of any given tree. We must then search over trees to find the tree of minimal cost.

Example: suppose we have four sequences AAG, AAA, GGA and AGA. Consider the two trees given below.



The number of mutations on each branch is shown to the left of the branch. The tree topology on the left has requires 3 mutations to explain the given sequences, while the tree on the right requires 4 mutations to explain the same sequences. The more parsimonious tree is therefore the one on the left. The sequences given at the internal vertices and the positions of the mutations could be altered in these examples but the total parsimony score for each tree would remain the same. □

An algorithm to compute the parsimony cost of a tree is given below. This finds the minimum number of substitutions to explain given sequences and tree. It assumes that all changes have equal cost. A similar algorithm accounts for the case where different substitutions have different costs. The algorithm is given in terms of rooted trees but the parsimony score is independent of the position of the root so this algorithm applies to unrooted trees.

Parsimony (Fitch 1971)

Number the nodes, in descending order, so that the root node is $2n-1$. Let u be the site for which we are considering the cost. Let B be the parsimony cost.

Initilise Set $B_u = 0$ and $k = 2n-1$.

Recursion To obtain the set R_k:

If k is a leaf node: Set $R_k = x_u^k$.

If k is not a leaf: compute R_i and R_j for child nodes of k. Set $R_k = R_i \cap R_j$ if $R_i \cap R_j \neq \emptyset$. Otherwise, set $R_k = R_i \cup R_j$ and set $B_u = B_u + 1$.

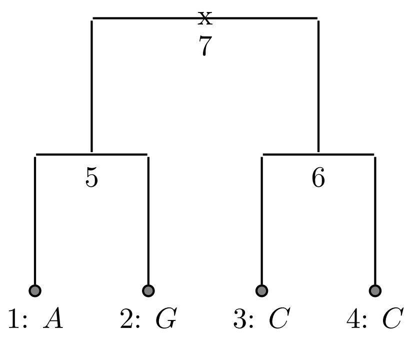
Stop Return B_u, the minimal cost of the tree at site u.

A traceback procedure can be used to construct possible ancestral states. Starting at the root, choose a residue from R_{2n-1} and go to the daughter nodes. Having chosen a residue at R_k, pick the same residue from the child set R_i if possible, otherwise choose a random reside of R_i.

The total cost for a tree and sequences is the sum of costs over all positions in the sequence. That is, if we have sequences of length L and B_i is the parsimony score for site i, then the total parsimony score for the tree and sequences is

$$B = \sum_{i=1}^{L} B_i.$$

Example: Given the following tree with just a single site at the 4 leaves we want to calculate the parsimony cost. Label the nodes as shown. There is just a single site so set $u = 1$.



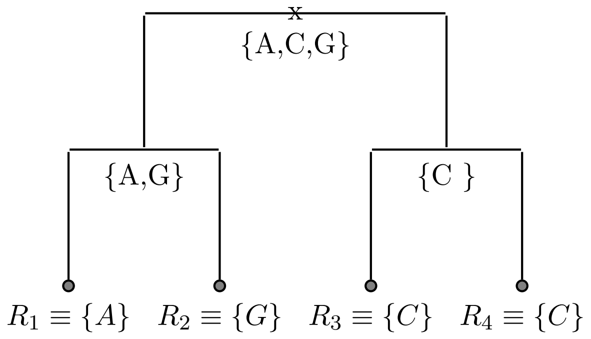
We set $B = 0$ and $k = 7$. Now try to find R_7.

7 is not a leaf node, so recurse down to its children. Want to find R_5 and R_6. 5 is not a leaf node so recurse down to its children. 1 is a leaf node so set $R_1 = \{A\}$. Similarly, $R_2 = \{G\}$. Now, $R_1 \cap R_2 = \emptyset$ so we set $R_5 = R_1 \cup R_2 = \{A, G\}$ and $B = B+1 = 0+1 = 1$.

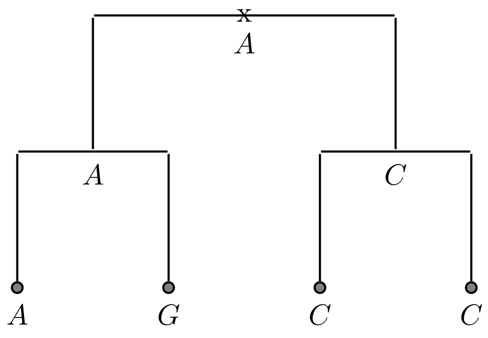
In a similar manner we get $R_3 = \{C\}$ and $R_4 = \{C\}$ so $R_6 = R_3 \cap R_4 = \{C\}$.

Now, since $R_5 \cap R_6 = \emptyset$ we set $R_7 = R_5 \cup R_6 = \{A, C, G\}$ and $B = B + 1 = 2$.

Thus we have the sets R_k as follows:



From these sets, we can traceback from the root, picking possible ancestral states that would give us the parsimony score for the tree. For example, at the root choose A, which forces us to choose A at node 5. Clearly, at node 6, we only have the choice of C.



□

20.5.1 Weighted parsimony

The basic parsimony idea easily extends to the case where instead of counting each mutation equally, different costs apply to different mutations.

Let $S(a, b)$ be the cost of mutating from a to b and again calculate the parsimony score at a single site u.

Initilise Set $k = 2n - 1$.

Recursion If k is a leaf node: Set $S_k(a) = 0$ when $a = x_u^k$ and $S_k(a) = \infty$ otherwise.

If k is not a leaf: Compute $S_i(a)$ and $S_j(a)$ for all a and children i and j of k. Set

$$S_k(a) = \min_b \left(S_i(b) + S(a, b)\right) + \min_b \left(S_j(b) + S(a, b)\right).$$

Stop Return

$$B_u = \min_a S_{2n-1}(a).$$

The total cost of the tree is

$$B = \sum_{u=1}^{L} B_u$$

Weighted parsimony reduces to the standard parsimony algorithm when $S(a, a) = 1$ and $S(a, b) = 0$ if $a \neq b$.

A traceback procedure to recover the ancestral states is again available by keeping track of which residue, b, gave the minimum at each step. For exact details of the traceback, see Durbin et al.

20.5.2 Parsimony informative sites

Many sites in an alignment will have the same parsimony score on every tree. For example, consider sites that have the same residue for all taxa (an invariant site). This will have a parsimony score of 0 regardless of the tree. Sites that have different scores on different trees are known as *parsimony informative.* It is easy to show that parsimony informative sites have at least two characters that each occur in two or more taxa.

In the detailed example given above, the site studied (which would be written as the column AGCC in a multiple sequence alignment for the 4 taxa) is not parsimony informative since there is only one site that appears more than once. If we instead considered the site corresponding to the column $AACC$, it would be parsimony informative.

20.6 Finding the maximum parsimony tree

The number of substitutions on a tree (the parsimony score) is sometimes called the *length* of a tree. This corresponds to the molecular clock idea where, under a constant rate mutation model, we will only see more substitutions if we wait for a longer time.

Thus finding the maximum parsimony tree is equivalent to finding the shortest tree. We'll consider a number of methods for finding the maximum parsimony tree for a given set of sequences.

20.6.1 Exhaustive search

Finding the maximum parsimony tree is a very hard problem computationally. Naive methods which attempt to score all possible unrooted trees fail when the number of sequences is even moderate due the huge number of possible trees.

We therefore need to resort to more clever methods and heuristic search algorithms.

The simplest of the smarter search algorithms are based on the idea of branch-and-bound.

(The following section of notes on parsimony is based on notes from `http://www.fos.auckland.ac.nz/~biosci742/4_3_2.html#4.3.4`)

20.6.2 Branch and bound

Branch and bound is a method of systematically analysing all possible trees by building up a tree one taxon (leaf) at a time and only continuing to build up a tree if it could potentially lead to the best tree.

Given n taxa, build an initial tree, t^* using some method. The score of that tree is s^*. Now we begin to systematically build up trees one taxon at a time as follows:

Initialise: Choose 3 taxa and form the (unique) unrooted partial tree.
Add this tree to a queue.

Iterate: Choose a taxon and add to previous best partial tree (at front of queue) in each possible position to get a k new partial trees, $t_1, \dots, t_k$
If $\mathit{score}(t_i) \leq s^*$, add t_i to queue and order the queue by score.
If $\mathit{score}(t_i) > s^*$, discard t_i.
If t_i is complete (all taxa have been added) and $\mathit{score}(t_i) < s^*$, set $s^* = \mathit{score}(t_i)$.

Finish: When queue is empty, return tree with lowest score.

This becomes clearer by looking at an explicit example so refer to Figure 19.

The result is effectively the same as an exhaustive search, without wasting time on topologies that we know will be rejected.

The algorithm can be optimised by having having a good initial tree (try perhaps using a neighbour-joining tree) and by ordering the taxa so that they are added in a way that promoter earlier cutoffs.

This is an improvement over exhaustive search (which is feasible for up to about 10 sequences) and is feasible for around 20-30 sequences.

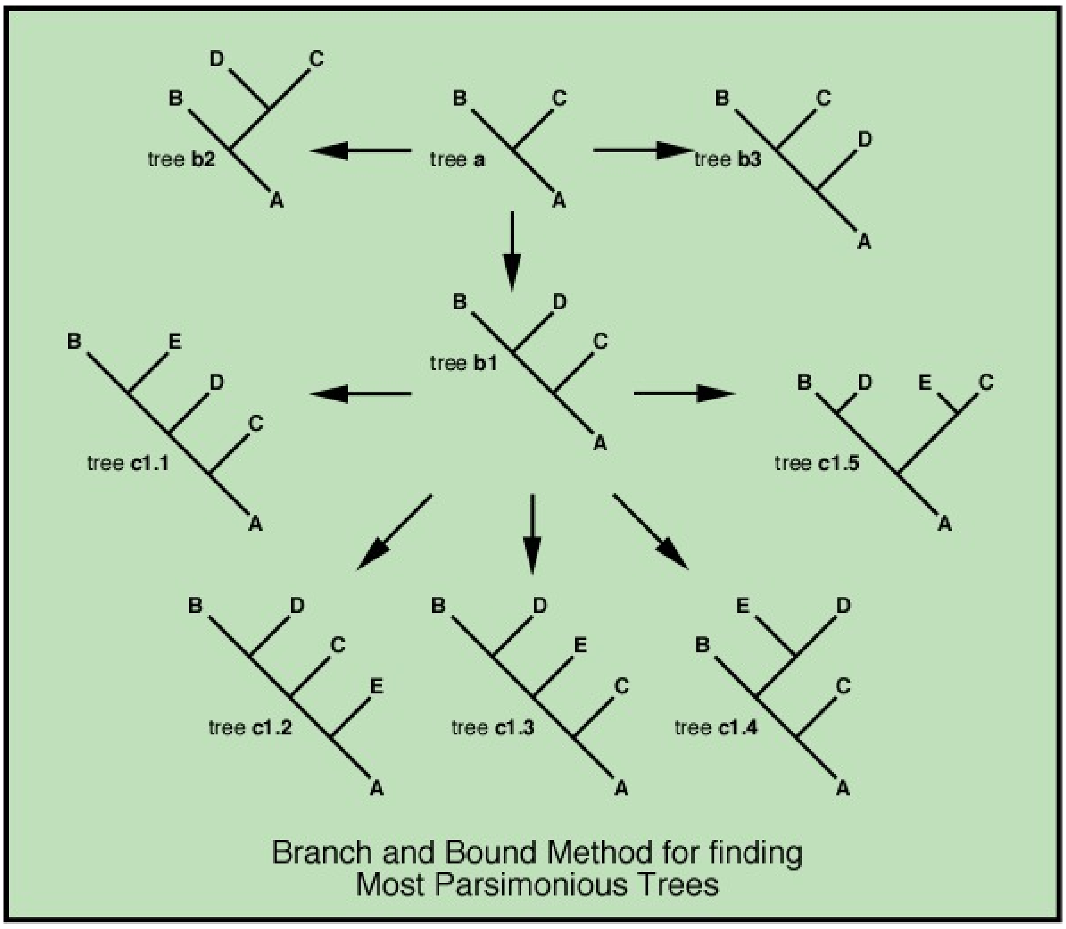


Figure 19: Say we have sequences from 5 taxa. We start by building the single 3-taxon tree using taxa A, B and C (tree a). Next the fourth taxon D is added in all three possible positions to generate trees b1, b2, and b3. One of these trees, say b1, is chosen. Then the fifth taxon E is added in all possible positions to give trees c1.1, c1.2, c1.3, c1.4, and c1.5. The length of each of these five 5-taxon trees is calculated. The shortest of these is the most parsimonious found to this point. Now return to the partial tree b2. If the length of b2 is equal to, or greater than, that of the shortest seen so far, then we know that adding any more taxa will only make the tree longer. If this is the case, then we stop using b2, and don't consider any of the trees built upon it. If b2 is shorter than the best seen so far, then it is used as the basis of further tree building, until the threshold length is reached. As we work through new topologies, we continuously update our record of the shortest seen so far. Once we have exhausted all possibilities, the shortest tree is the most parsimonious for that alignment.

20.6.3 Heuristic search

Heuristic methods search for the optimal tree but offer no guarantee that it will be (or has been) found. These methods use hill-climbing to seek the optimal tree:

- choose an initial tree
- Iterate:
  - modify the tree and assess it
  - if the modified tree is an improvement, keep it. Else, return to the previous tree
  - stop when no improvement occurs

The initial tree can be chosen using *stepwise attachment*, a greedy algorithm that starts by joining 3 taxa into a tree and then progressively adds further taxa by finding the best place to attach a taxon and leaving it there. Since taxa are never moved once they have been attached even if it becomes obvious that something has been attached in the wrong place, this method will almost never find the best tree to start with. It will, however, nearly always give us something better than the worst tree.

Modifications to the tree can be made by various methods of detaching and reattaching branches in different a different place. This is known as branch swapping. An example of one type of re-arrangement method is given in Figure 20.

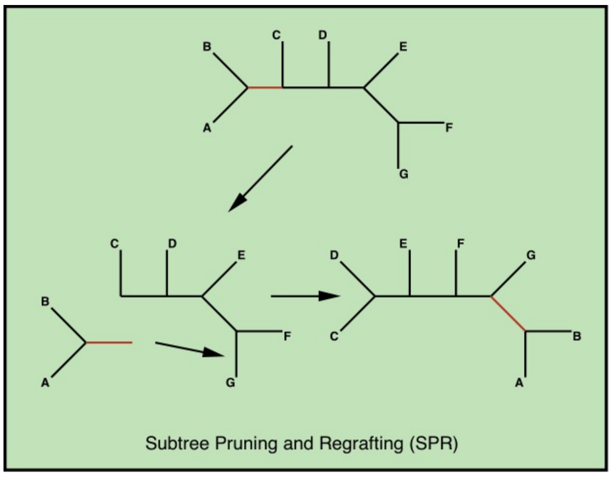


Figure 20: An example of a way of modifying the tree: subtree pruning and regrafting. An edge is chosen and the subtree at that point is removed. Another edge is chosen on the remainder of the tree and the removed subtree is reattached at that point.

The method described above is guaranteed to find a local minimum of the parsimony score, but may not find the global minimum as the starting tree may be too far from the best tree.

To improve the chances of finding the best tree, the method for building the initial tree can be randomised so the the starting point is different in every case. For example, if the initial tree is built by stepwise attachment, the order in which the taxa are added can be randomised. Different starting points may end up finding different local minima.

20.7 Disadvantages of parsimony

Beyond the difficulty of finding the maximum parsimony tree, parsimony has several disadvantages.

Firstly, parsimony does not account for hidden or multiple substitutions at the same site as it explains all substitutions with the minimum possible number of changes. So if we observe a locus in three sequences, one with an A, the other two with a C. Reconstructing the parsimony tree, we will assume that the ancestral state was a C and a single mutation had occurred to produce the A. There are clearly many other explanations for this data set (for example, there were multiple mutations so that the ancestral state was A and two mutations to C occurred or the was a hidden mutation where the ancestral state was a C, there was a mutation to a G and then an A) which, although each less likely than the single mutation, collectively are quite likely. This effect means that *parsimony tends to underestimate the length of trees.*

The most serious problem with maximum parsimony is *long-branch attraction*, a consequence of the failure of the method to estimate multiple or hidden substitutions. When a tree has some branches with significantly greater length than other branches, MP will underestimate how many substitutions have occurred on the long branches. Homoplasies (parallel or convergent substitutions) will cause MP to underestimate the evolutionary difference between the branch tips. Conversely it will over-estimate the degree to which those tips have shared an evolutionary past. The long branches will be joined erroneously as near- or sister-clades, that is they will "attract" one another. Using longer genomic sequences in the analysis will only increase the number of variable sites exhibiting homoplasies, without improving the phylogenetic signal. As a consequence of this, MP is statistically inconsistent, that is, the chance of obtaining the wrong answer increases as more data are used. It can be positively misleading. See Figure 21.

So while parsimony is simple to understand as a heuristic, relatively quick to implement and compute and will do a good job of reconstruction when substitutions are rare, it does not explain the process of sequence evolution well (no physical model of the process) and is prone to failing when there are hidden and multiple substations, especially when there are long branches (highly diverged sequences) in the tree. A further problem is that the maximum parsimony tree is just a single tree that contains no information about uncertainty — we aren't sure which splits in the tree we are certain about or which splits could be rearranged to produce an equally likely (or very near to equal) tree.

We thus seek a method of reconstructing a tree that is based on statistical principles, one that will find the most likely tree taking into account a model of the process that gave rise to the data. The method should also give us an idea of the uncertainty in the

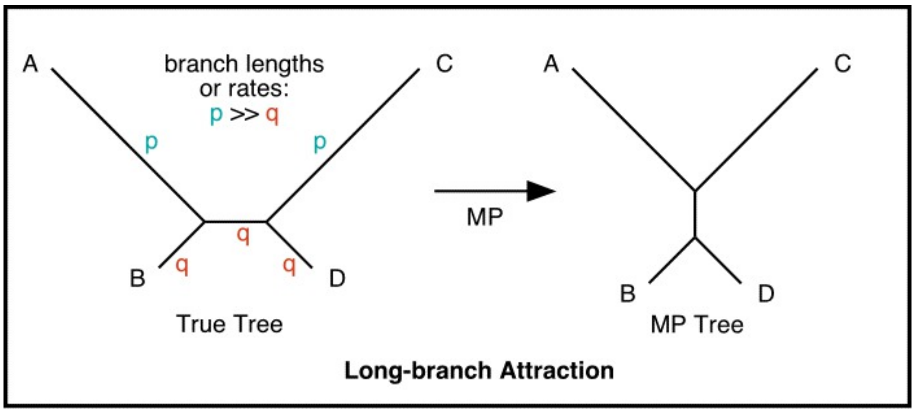


Figure 21: The tree on the left is the true tree. It has a pair of sequences (A, and B) which are highly diverged. Reconstructing the tree from these sequences using Maximum Parsimony (MP) results in the wrong tree (right). The two highly diverged sequences look closer to each other than they should due to chance mutations.

reconstruction. Indeed, we will look at methods that provide us with many different possible trees that all represent feasible reconstructions of the evolutionary relationships between sequences.

21 Statistical approaches to modelling evolution

Distance and parsimony based methods for tree reconstruction are based on a number of assumptions that often do not hold. Distance methods are simple and fast to implement but are only guaranteed to reconstruct the correct tree under very restrictive circumstances. Parsimony is not based on a realistic model of evolution and, as we have seen, is statistically inconsistent (it reconstructs the wrong tree even with infinite data).

Our approach then will be similar to the approach we have taken in other parts of the course: we will model the process of sequence evolution, and based on that model we will write down the likelihood of a tree. We will then seek to find the maximum likelihood tree and finally look at Bayesian approaches to finding the best tree.

We model only the *substitution process* in which one base is replaced by another, for example $A \to T$ or $A \to C$. We will ignore the (more complicated) processes of insertions, deletions, recombination etc.

21.1 Likelihood of a given tree

Consider a tree with four leaves and sequence at each leaf consisting of a single site. An example of such a tree with four sequences, labeled A, B, C, and D is shown in Figure 22. The values of the sequences are C, C, T and T, respectively.

The maximum parsimony tree for these sequences groups A and B together and has a parsimony score of 1. But how likely is it? Inherent in the parsimony idea is that only one mutation occurred on the tree and it must have occurred along the branch between the two ancestral nodes. That mutation was from a C to a T (or vice-versa) implying that the unknown ancestral values at the ancestral nodes were also C and T, as shown on the left in Figure 22.

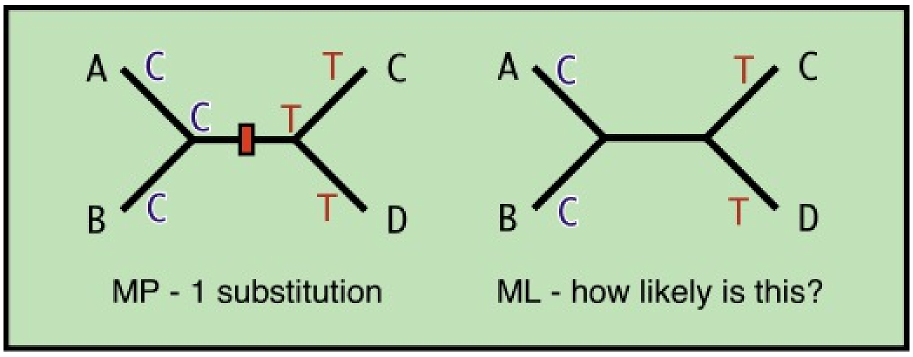


Figure 22: The parsimony tree on the left with the ancestral states reconstructed. Under parsimony, this is considered to be the one true tree. Under likelihood methods, we want to decide how likely the tree is given the data (observed values at the leaves). That requires summing over all possible ancestral values (shown in 23).

Ideally, we would account for all possibilities for the ancestral values: they could be any of $(A, A), (A, C), \dots (T, T)$. All possibilities are shown in Figure 23. That is, if the unknown ancestral states are X and Y, then we could look at the likelihood of the tree for each possible combination of X and Y and sum these together. This is the principle of marginalisation introduced in Section 11.4: we want to know the probability of the tree and the data, but have some other random variables floating about too (the ancestral states X and Y) which we deal with by simply summing over all possible values to get

$$P(\text{Tree and data}) = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} P(\text{Tree and data}, x, y)$$

It turns out that once we have a tree with values for the site known at all nodes (not just the leaves), we can calculate the likelihood with relative ease. That is, we know how to calculate $P(\text{Tree and data}, x, y)$ for any value of x and y, a sketch of which is given in Figure 24.

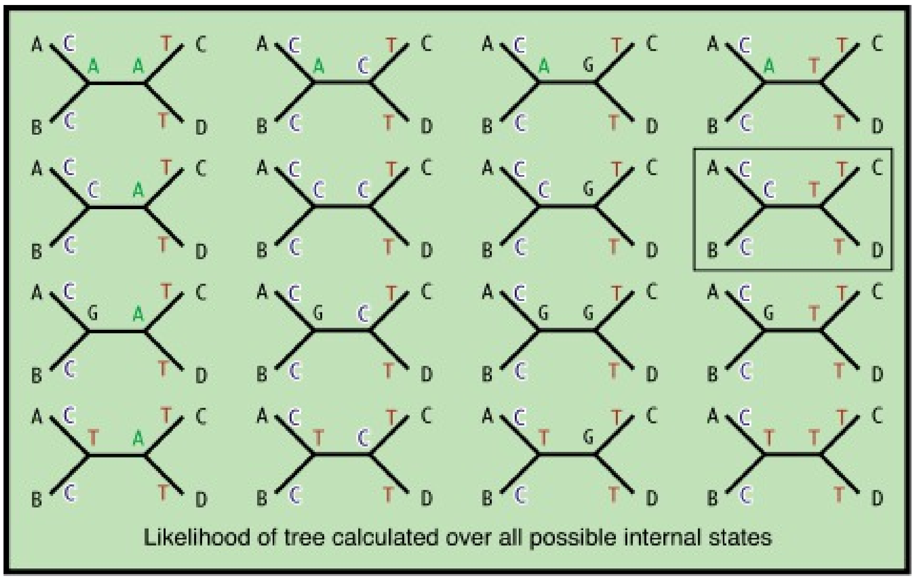

Likelihood of tree calculated over all possible internal states

Figure 23: All possible ancestral value for the tree considered in Figure 22. The likelihood of the tree is the sum of the likelihoods of the tree and the ancestral values, where the sum is over all ancestral values. That is, to find the likelihood of the tree in on the right in Figure 22, we need to find the likelihood of each of the trees in this figure and sum them up.

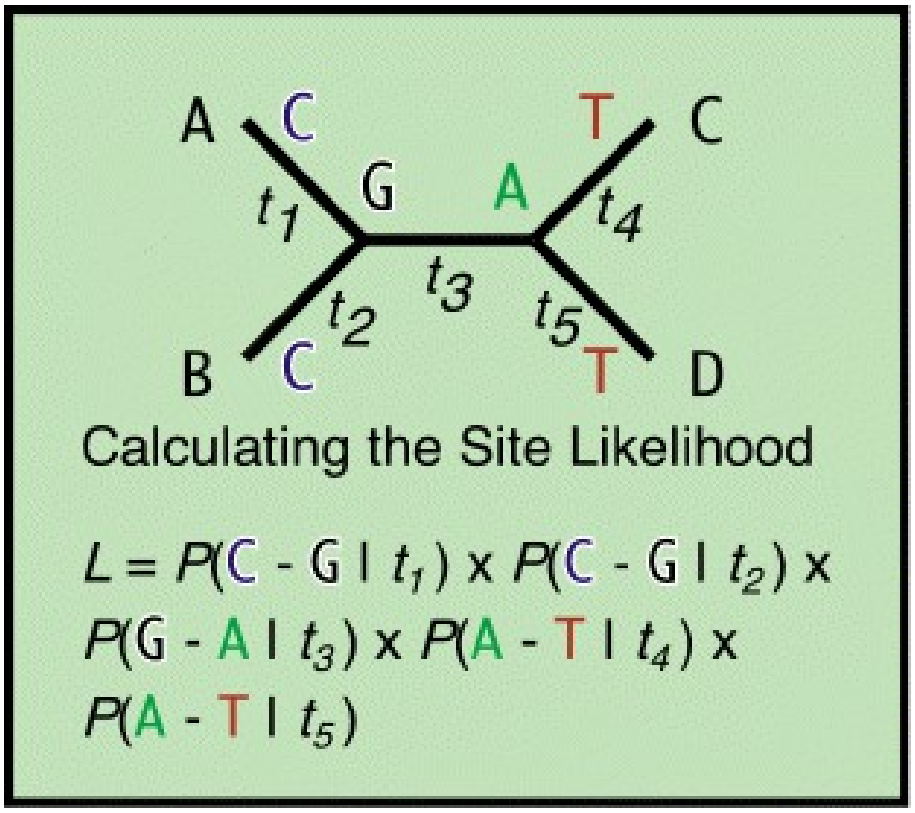

A
C
T
C
G
A
t_1
t_4
t_3
t_2
t_5
B
C
T
D
Calculating the Site Likelihood
$L = P(C - G \mid t_1) \times P(C - G \mid t_2) \times$
$P(G - A \mid t_3) \times P(A - T \mid t_4) \times$
$P(A - T \mid t_5)$

Figure 24: The likelihood for a tree with data at the leaves and imputed ancestral data is given by a product of the probabilities of mutating between the different values along each branch: that is the probability of mutating from C to G along the branch of length t_1 multiplied by the probability of mutating from C to G along the branch of length t_2 multiplied by the probability of mutating from G to A along the branch of length t_3 and so on. How these probabilities are calculated in developed in Section 21.2

Let's formalise the discussion above.

The data we have about a tree can be considered as the matrix D, where D_{ij} is the value of the jth residue in the ith individual. For a given data set, D, we want to calculate the likelihood of a given tree, T, which is binary and has a leaf associated with each individual (each row of the matrix).

So we want $\mathcal{L}(T) = P(D|T)$. We saw that model each site as being independent of all other sites. That is, we consider

$$P(D|T) = \prod_{i=1}^{L} P(D_{:,i}|T).$$

But we saw that to calculate $P(D_{:,i}|T)$, we need to sum over all possible unknown values for the sites at the internal nodes. For a given tree, let $\mathbf{A}$ be the matrix of unknown ancestral sequences at each node of the tree, where each row corresponds to an ancestral node and each column to a site.

Then for each column i,

$$P(D_{:,i}|T) = \sum_{A_{:,i}} P(D_{:,i}, A_{:,i}|T).$$

That is, for each site we sum of all possible assignments of ancestral values to get the likelihood for that site. It turns out that this sum over all ancestral values is easy enough to calculate using a dynamic programming approach, so we just need to figure out how to calculate each element of the sum: $P(D_{:,i}, A_{:,i}|T)$.

We make one further assumption: that each lineage is independent. In that case, the probability of the tree with states known at every node, $P(D_{:,i}, A_{:,i}|T)$, becomes a simple product of the likelihoods of each lineage of the tree.

That means all we need is a model that gives us the probability of mutating from base a to base b over an arbitrary length of time (which is given by the branch length).

We have omitted one detail here, and that is the probability of observing the sequence of the most recent common ancestor of the full sample.

The schematic of this argument is given in Figure **??**.

We introduce the model we use for mutations down each lineage in next Section and also briefly discuss how the sum $\sum_{A_{:,i}} P(D_{:,i}, A_{:,i}|T)$ can be calculated efficiently.

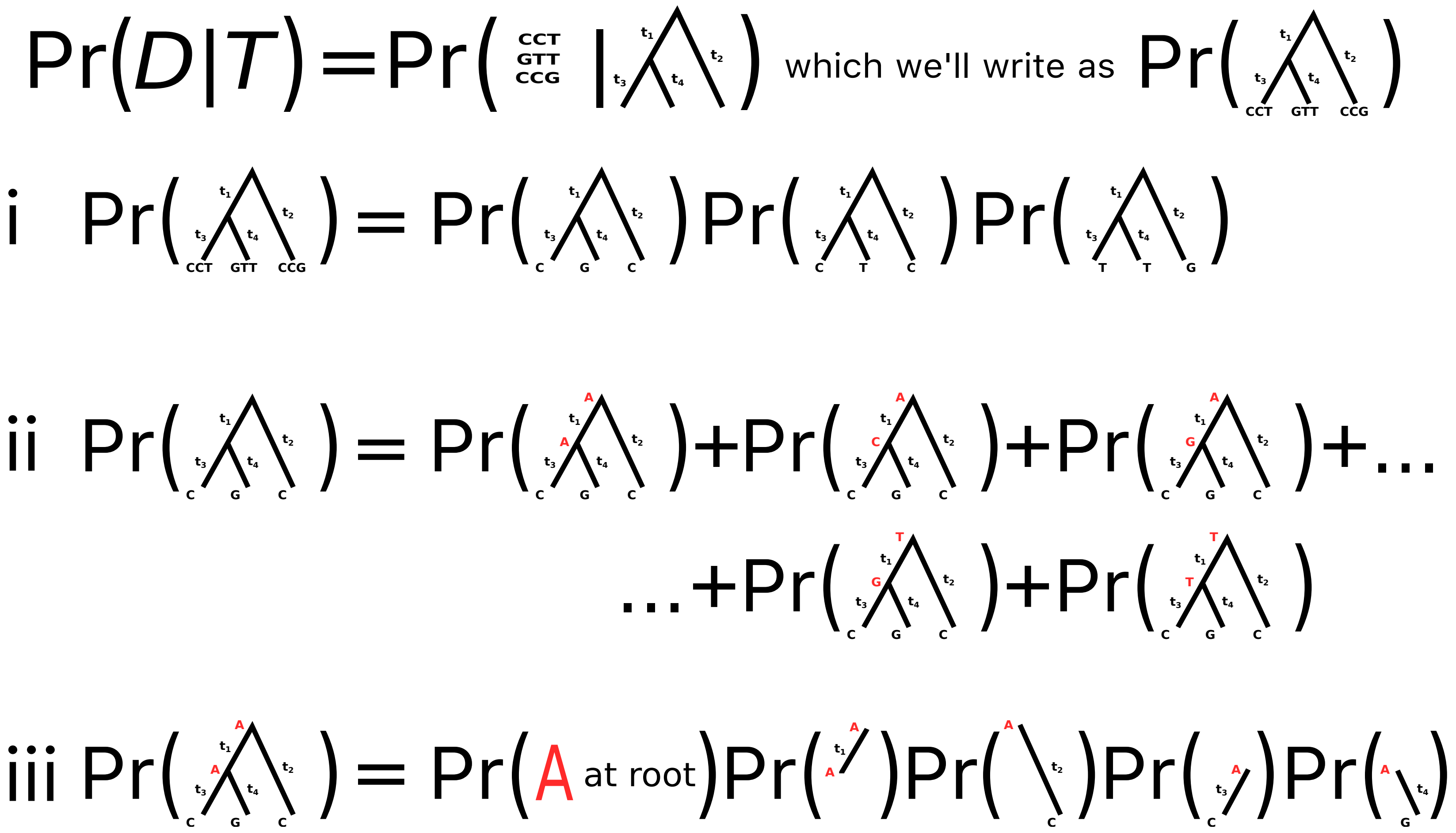

Pr(D|T)=Pr(CCT GTT CCG | tree) which we'll write as Pr(tree CCT GTT CCG)
i Pr(CCT GTT CCG) = Pr(C G C)Pr(C T C)Pr(T T G)
ii Pr(C G C) = Pr(A A)+Pr(A C)+Pr(A G)+...
...+Pr(T G)+Pr(T T)
iii Pr(A A C G C) = Pr(A at root)Pr(t1)Pr(t2 C)Pr(t3 C)Pr(t4 G)
t1
t2
t3
t4

Figure 25: A schematic representation of calculating the likelihood of a tree $P(D|T)$. Equation i holds since each site (column of D) is assumed independent, so $P(D|T) = \prod_{i=1}^{L} P(D_i|T)$. Equation ii expands the first term on the right of equation i. It shows all possible ancestral values (shown in red) being summed over, $P(D_i|T) = \sum_{A_i} P(D_i, A_i|T)$. The sum, here containing 16 terms, can be efficiently calculated using dynamic programming. Equation iii expands the first term on the right of equation ii and relies on an assumption of independent evolution down different lineages. The probability of the root value is given by the stationary distribution π, while the probability of having X at the top of a lineage of length t and Y at the bottom is given by $P_{XY}(t) = [\exp(tQ)]_{XY}$. So the right of equation iii equals the product $\pi_A P_{AA}(t_1) P_{AC}(t_2) P_{AC}(t_3) P_{AG}(t_4)$.

21.2 Markov processes

The model we use is a continuous time Markov model (often called *Markov processes*). We haven't got time to go into this in much detail but the main idea behind it is relatively intuitive.

We've already seen a simple Markov process, in the form of the Poisson process. Recall that in the Poisson process, 'events' can occur at any time, times between events are exponentially distributed and, in a very period of time, we expect either to see no events or one event (but not two or more). In the Poisson process, we only considered a single type of event and simply counted them when they occurred so that the state of the process was just a count of the number of things that had occurred so was strictly increasing.

The model we use for mutations is similar to the Poisson process in that waiting times between events are exponential and, in a short slice of time, nothing or just one event will occur. The events are substitutions so we want to keep track of the state of the process at each time point. Let $X(t)$ the state of the process at time t (so $X(t) \in \{A, C, G, T\}$). The difference from a Poisson process is that we allow that the rate of different events occurring may be different. For example, the rate of mutations from state A to state G may be different from the rate at which A mutates to T.

Let q_{ab} be the instantaneous rate of transitions from state a to state b, for any states $a \neq b$. It may help to think of this in terms of flow: q_{ab} is the rate of flow form state a to state b.

q_{aa} is the total rate of flow out of a, so it is the sum of rates from a to b for $b \neq a$ and is defined to be negative, $q_{aa} = -\sum_{a \neq b} q_{ab}$. This means that the length of time spent in state a is exponentially distributed with rate $-q_{aa}$.

Now define the rate matrix Q to have off diagonal elements q_{ab} and diagonal elements $q_{aa} = -\sum_{a \neq b} q_{ab}$.

In the case of modelling DNA mutations, Q has the form

$$Q = \begin{pmatrix} -\sum_{j \neq A} q_{Aj} & q_{AC} & q_{AG} & q_{AT} \\ q_{CA} & -\sum_{j \neq C} q_{Cj} & q_{CG} & q_{CT} \\ a_{GA} & q_{GC} & -\sum_{j \neq G} q_{Gj} & q_{GT} \\ q_{TA} & q_{TC} & q_{TG} & -\sum_{j \neq T} q_{Tj} \end{pmatrix}.$$

where q_{ij}, $i \neq j$ can be interpreted as the instantaneous rate that i mutates to j.

Now define $P_{ab}(t) = \Pr(X(t) = b | X(0) = a)$ the probability of starting in state a at time 0 and being in state b at time t and form the matrix $P(t) = [P_{ab}(t)]$. It turns out that we can derive $P(t)$ from Q by taking the matrix exponential:

$$P(t) = \exp(Qt) = \sum_{k=0}^{\infty} \frac{(Qt)^k}{k!}.$$

The matrix exponential can sometimes be calculated analytically but, for general Q, use numerical methods which are available for many standard numerical libraries.

The matrix exponential follows the rules we would hope for the exponential function. For example:

- $\exp(\mathbf{0}) = \mathbf{I}_n$
- when $\mathbf{AB} = \mathbf{BA}$, $\exp(\mathbf{A} + \mathbf{B}) = \exp(\mathbf{A}) \exp(\mathbf{B})$
- When $\mathbf{B}$ is invertible, $\exp(\mathbf{BAB}^{-1}) = \mathbf{B} \exp(\mathbf{A}) \mathbf{B}^{-1}$
- If $\mathbf{A} = \operatorname{diag}(a_1, \ldots, a_n)$, then $\exp(\mathbf{A}) = \operatorname{diag}(\exp(a_1), \ldots, \exp(a_n))$.

21.3 Models of sequence mutation

There are a lot of parameters to specify or estimate in a full model of sequence mutation: on the face of it, we need to specify each of the q_{ab}s (giving us 12 parameters when there are 4 bases). In reality, we make various simplifying assumptions to reduce the number of parameters and make estimation simpler.

We insist that models of mutation are *reversible* and *stationary.*

A process is stationary when , if run for long enough, it settles down to some equilibrium, $\pi = [\pi_A, \pi_C, \pi_G, \pi_T]$. Formally, π is defined by the equation $\pi P(t) = \pi$ for any t. That is, if the process starts in equilibrium, running the process further will leave it in equilibrium. π is called the equilibrium distribution or the stationary distribution.

A process is reversible when it looks the same running backwards as it does running forwards. Strictly, a process is reversible when it satisfies the detailed balance conditions $\pi_i p_{ij}(t) = \pi_j p_{ji}(t)$ for all values of i, j and t.

It helps when specifying substitution matrices to normalise them so that the average number of mutations per unit time is one. Given an unnormalised matrix $\hat{Q}$, it can be normalised by multiplying each entry by some constant β, so that the normalised matrix is $Q = \beta\hat{Q}$. The value of β depends on $\hat{Q}$.

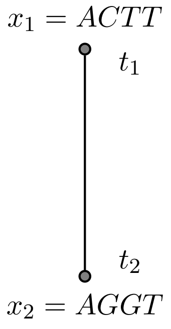
Scaling the substitution matrix in this manner means that time is measured in substitutions. That is, one time unit corresponds to the time in which we would expect to see one mutation at any given site.

Summary of assumptions we make

- Each site is identical to all others in the evolutionary processes operating on it.
- Each site is free to change independently of all other sites.
- These two assumptions are usually stated as sites are independent and identically distributed, or i.i.d.

- Substitution probabilities do not change with time or over the tree. This is known mathematically as a homogeneous Markov process.
- The mutation process is time-reversible meaning that the process looks the same whether it is run forward or backward in time.
- Mutation process are independent across across different branches.

So given a mutation model (that is, given Q and μ), we can easily determine the probability of observing a descendant sequence given an ancestral sequence. For example, given the sequence $x_1 = ACTT$ at time t_1 and the $x_2 = AGGT$ at time t_2 and assuming



But given a tree with samples at the tips, the ancestral sequences are unknown to us. But how does this work when we don't know the ancestral sequences? We need to sum over all possible ancestral sequences. It turns out that this is relatively easy to achieve using a dynamic programming approach known as Felsenstein's peeling (or pruning) algorithm (1981) which performs this calculation in polynomial time based on a post-order traversal of the tree. We omit the details.

21.3.1 Jukes-Cantor model

The simplest model is the *Jukes-Cantor* model (1969) which has equal rates of mutation between all bases so that $q_{ij} = 1$ for $i \neq j$,

$$Q = \beta \begin{bmatrix} -3 & 1 & 1 & 1 \\ 1 & -3 & 1 & 1 \\ 1 & 1 & -3 & 1 \\ 1 & 1 & 1 & -3 \end{bmatrix}.$$

In this case, $\beta = 1/3$ so

$$Q = \begin{bmatrix} -1 & 1/3 & 1/3 & 1/3 \\ 1/3 & -1 & 1/3 & 1/3 \\ 1/3 & 1/3 & -1 & 1/3 \\ 1/3 & 1/3 & 1/3 & -1 \end{bmatrix}.$$

The equilibrium of this process is $\pi = (1/4, 1/4, 1/4, 1/4)$.

The transition matrix $P(t) = \exp(Qt)$ for the Jukes-Cantor model has off-diagonal entries

$$P_{ij}(t) = \frac{1}{4} - \frac{1}{4}\exp\left(-t\mu\right)$$

and diagonal entries

$$P_{ii}(t) = \frac{1}{4} + \frac{3}{4}\exp\left(-t\mu\right).$$

21.3.2 Kimura model

The Kimura model (1980) distinguishes between *transitions* ($A \longleftrightarrow G$ and $C \longleftrightarrow T$ state changes) and *transversions* (state changes from a purine to pyrimidine or *vice versa*). The model assumes base frequencies are equal for all characters. This transition/transversion bias is governed by the κ parameter and the Q matrix is:

$$Q = \beta \begin{bmatrix} -2-\kappa & 1 & \kappa & 1 \\ 1 & -2-\kappa & 1 & \kappa \\ \kappa & 1 & -2-\kappa & 1 \\ 1 & \kappa & 1 & -2-\kappa \end{bmatrix},$$

The normalized Q is obtained by setting $\beta = \frac{1}{2+\kappa}$. This model has one free parameter, κ. The transition probabilities are:

$$p_{ij}(d) = \begin{cases} \frac{1}{4} + \frac{1}{4}\exp(-\frac{4}{\kappa+2}d) + \frac{1}{2}\exp(-\frac{2\kappa+2}{\kappa+2}d) & \text{if } i = j \\ \frac{1}{4} + \frac{1}{4}\exp(-\frac{4}{\kappa+2}d) - \frac{1}{2}\exp(-\frac{2\kappa+2}{\kappa+2}d) & \text{if transition} \\ \frac{1}{4} - \frac{1}{4}\exp(-\frac{4}{\kappa+2}d) & \text{if transversion} \end{cases}.$$

21.3.3 F81 and HKY models

In 1981, Joe Felsenstein proposed a model that extends the Jukes-Cantor model to allow for unequal equilibrium base frequencies, that is π for which $\pi_a \neq \pi_b$. This is known as the F81 model. The F81 model has 3 parameters, one less than the number of equilibrium base frequencies since there is the restriction that $\sum_i \pi_i = 1$.

In 1985, the F81 model was extended to incorporate the Kimura model, so allows different rates for transitions and transversions as well as unequal base frequencies. The resulting model is known as the HKY model and has rate matrix of the form:

$$Q = \beta \begin{bmatrix} \cdot & \pi_C & \kappa\pi_G & \pi_T \\ \pi_A & \cdot & \pi_G & \kappa\pi_T \\ \kappa\pi_A & \pi_C & \cdot & \pi_T \\ \pi_A & \kappa\pi_C & \pi_G & \cdot \end{bmatrix}$$

where the diagonal elements are defined in the usual way so that the row sums are zero. The transition matrix P can be calculated analytically for this model but it is omitted here.

21.3.4 GTR model

In 1986, the most general reversible model was developed which can have an arbitrary stationary distribution, and given the restriction of reversibility, 6 parameters for adjusting the rates of mutation between bases. The rate matrix is

$$Q = \beta \begin{bmatrix} - & a\pi_C & b\pi_G & c\pi_T \\ a\pi_A & - & d\pi_G & e\pi_T \\ b\pi_A & d\pi_C & - & f\pi_T \\ c\pi_A & e\pi_C & f\pi_G & - \end{bmatrix}.$$

The diagonal elements are calculated in the normal way.

Where the normalization is $\beta = 1/[2(a\pi_A\pi_C + b\pi_A\pi_G + c\pi_A\pi_T + d\pi_C\pi_G + e\pi_C\pi_T + \pi_G\pi_T)]$

This model has 9 parameters to be specified: the parameters of the equilibrium distribution, $\pi = (\pi_A, \ldots, \pi_T)$, (since $\sum_i \pi_i = 1$, this only counts for 3 parameters) and the parameters $a, b, c, d, e, f > 0$. Note the form of the Q matrix here is chosen so that π is indeed the equilibrium distribution, that is, as $t \to \infty$, every row of $P(t) \to \pi$. Recall that $P(t) = \exp(tQ)$, where exp() is the matrix exponential.

The same modelling tools can be used when the bases are the 20 amino acids, the difference being that the Q matrix is now 20×20.

21.3.5 Rate variation across sites

An obvious property many real alignments share is that different sites (i.e., columns) in the alignment appear to mutate at different rates: some sites appear to essentially be at equilibrium (so bases in that column are close to what you would expect if they were randomly drawn from the stationary distribution) while other sites may be constant.

To account for this, multiply the rate matrix for each site by some multiplier r_i, $i = 1, \ldots, L$, where L is the number of sites. Typically, the r_is are modelled as coming from a gamma distribution with mean 1 and variance determined by the shape parameter α. To simply computation, the gamma distribution is discretised into K categories (that is, it is approximated by a discrete distribution taking K possible values), so that r_i can only take one of K possible values. The r_i are known as relative rates. Seeing the mean of the r_is is 1, the expected number of mutations per unit time per site is still 1.

Finally, the constant sites in the alignment can be dealt with directly by allowing the the relative rate to be zero at those sites.

The shorthand used for these models is, for example, GTR+Γ+I, meaning that the general time reversible model is used with rate variation treated under the (discrete) gamma model with invariant sites allowed for.

21.4 Estimating the maximum likelihood tree

According to the substitution model we are using, the best tree is the one which maximises the likelihood $\mathcal{L}(T) = \Pr(D|T)$ under that model. This is called the maximum likelihood tree. Since there is no way to analytically find the maximum likelihood tree under general model of mutation, we can use similar techniques to those used for maximum parsimony to find something close to the maximum likelihood tree.

That is, we can start at some tree and use a stochastic search to propose new trees which are accepted if they have a higher likelihood. Note that we have the added complication when dealing with likelihoods that branch lengths now influence the likelihood of a tree, so for each tree topology, the branch lengths need to be optimised.

The hill-climbing algorithm we introduced in the context of parsimony trees is restated here for likelihood trees:

- choose an initial tree and calculate its likelihood.
- Iterate:
  - modify the tree and calculate its likelihood.
  - if the modified tree has a higher likelihood then the unmodified tree, keep it. Else, return to the previous tree
  - stop when no or minimal increase in likelihood occurs

Modifications to the tree can either change the tree topology (shape) or the length of the branches. The same topology changing operations as we used in the equivalent parsimony algorithm, such as SPR, can be extended to work with trees with explicit branch lengths as we have here. Modifications that change only the branch lengths are also used in this context.

21.5 Bayesian approach to phylogenetics

What we really want is not just the likelihood, but the posterior probability of the tree (and other model parameters) given the data. That is, given data D we want to find the posterior distribution $P(g, Q, \mu|D)$. From Bayes' Theorem,

$$P(g, Q, \mu|D) = \frac{P(D|g, Q, \mu)P(g, Q, \mu)}{P(D)}$$

where $P(D|g, Q, \mu)$ is the likelihood for the parameters with fixed data, $P(g, Q, \mu)$ is the prior distribution for the parameters and $P(D)$ is a normalisation constant.

Once again, we are unable to calculate this analytically so resort to numerical techniques to study this object. The primary computational tool that is used in Bayesian phylogenetics is Markov chain Monte Carlo (MCMC). MCMC gives us a method of generating samples from the posterior distribution. These samples form the basis of our investigation of the posterior distribution.

21.6 Models for trees: Yule trees and the coalescent

We look at two basic models for trees, the Yule tree and the coalescent. Yule trees are typically used when we have observed sequences from multiple different species. It models speciation as rare, random events. The coalescent is used to model trees when all sequences came from the same population and species. It is based on a model of how individuals in a population interact.

21.6.1 Yule trees

A Yule tree can be viewed as a realisation of the Yule process which is a pure both process. It starts at time $t_0 = 0$ with one lineage (species). Each lineage branches according to a Poisson process with rate λ. That is, a lineage branches after some time $t \sim Exp(\lambda)$. When a lineage branches it produces an exact copy of itself that proceeds in the same way and will go on to branch according to this same process. Here, branching model speciation events, where all species are equally likely to speciate.

So when there are k lineages, branching occurs at according to Poisson process with total rate $k\lambda$.

We usually ignore the branch above the root, so start the process with $k = 2$. So to simulate the Yule process, set $k = 2$ and $t = 0$ and make a root node at time t with two lineages. Iterate: Draw $t_k \sim Exp(k\lambda)$ and set $t = t + t_k$. Pick one of the lineage uniformly at random by copying it by making a node with time t and two children. Increment k.

To produce a tree with n leaves, stop the process immediately before the $n+1$th lineage is produced (at time $\sum_{k=2}^{n} t_k$.

We could have produced this tree in reverse by starting with n leaf nodes and n lineages above them. Let $k = n$. Simulate a time $t_k \sim Exp(k\lambda)$ and set $t = t + t_k$. Choose a pair of lineages uniformly at random. Make a node at time t with these two chosen lineages as children. Decrement k. Stop when $k = 1$.

Given a tree, g, that was produced according to this process we can write down the probability density of this tree. Label the nodes with time increasing back into the past, so leaf nodes to have time $t_n = 0$ and node k has time t_k and there are k lineages between time t_k and t_{k-1}. Then the probability density function of a tree, given the branching rate λ is,

$$f(g|\lambda) = \prod_{k=2}^{n} \lambda \exp\left(-k\lambda(t_{k-1} - t_k)\right).$$

21.6.2 The coalescent

The coalescent comes from considering how two individuals in a simple population are related. The population is modelled by one of the simplest models of a interacting population used in population genetics, known as the Wright-Fisher model. The Wright-

Fisher model has a constant size population consisting of N individuals and generations are discrete. All individuals are equally likely to produce offspring in the next generation. That is, at the end of a generation, the whole current population dies and is replaced by their offspring.

This can be modelled as thinking of each of the N offspring in the $(k+1)$th generation choosing a parent uniformly at random from the kth generation (that is, the previous generation). So, on average, each individual in the kth generation has one child surviving in the $k+1$th generation but some will have $0, 1, 2$, etc.

The coalescent comes about by considering the process going backward in time and asks, if we choose two individuals in the current generation uniformly at random, who many generations do we have to go back until this pair shares a common ancestor?

The chance that they share an ancestor in the previous generation is simply that the probability that they share a parent, which has probability $1/N$ (recall that N is the population size), while the probability they do not share a parent is $1-1/N$. If they don't share a parent, the same argument holds for each preceding generation: they share a parent with probability $1/N$ and not with probability $1-1/N$. When the pair share a parent, we say that they 'coalesce' in the parent generation. Let T be the number of generations back to coalescence. Then

$$\Pr(J=j) = \frac{1}{N}\left(1-\frac{1}{N}\right)^{j-1}$$

which is just a geometric distribution. So the expected time to coalescence is N generations. It is natural to chose a time scale so that one time unit is N generations and consider the limit as N gets large.

So set $t = \frac{j}{N}$ where j is time measured in generations. Now, $j = tN$ Then the probability that the two lineages have not coalesced in t time units is

$$\left(1-\frac{1}{N}\right)^{tN-1} \longrightarrow e^{-t} \text{ as } N \to \infty.$$

That is, the expected time to coalescence for a pair of individuals in a large population is exponentially distributed with parameter 1. The result is true for any pair, so if we choose k indidividalusl and trace their ancestry back, each pair coalesces at rate 1 and, since there are $\binom{k}{2}$ pairs, coalesences occur at total rate $\binom{k}{2}$ (that is the first coalescence between some pair will occur after an exponentially distributed time with parameter $\binom{k}{2}$). Remember that time was scaled so that 1 time unit was equal to N generations, so the time to coalescence is proportional to the population size.

Thus we have a method of simulating coalescent trees.

1. Set $k = n, t = 0$.
2. Make n leaf nodes with time t. This is the set of available nodes.
3. While $k \geq 2$, iterate:

4. Generate a time $t_k \sim \text{Exp}(\binom{k}{2})$. Set $t = t + t_k$.
5. Choose two nodes uniformly at random and let them coalesce at a new node with time t. Replace the two chosen nodes in the set of available nodes with this new node.
6. Set k = k-1.

A coalescent tree reflects the ancestry of a sample of n individuals drawn from a large population of N.

It is often convenient to measure time in units of expected mutations. Consider a Wright-Fisher model in which mutations occur with probability u at each generation. That is, there is a probability u that an individual differs from its parent by a single mutation. Let $\theta = 2Nu$, so θ is the expected number of mutations separating 2 sequences (since there are N generations back to a common ancestor, on average, each lineage picks up Nu, so there are a total of $2Nu$ mutations between the two lineages).

So if we want to define time in units of expected number of mutations, we multiply coalescent time (which is measured in units of N generations) by $\frac{\theta}{2}$, that is multiple all branch length by $\frac{\theta}{2}$. So instead of a rate of coalescence of 1 for each pair, we have a rate of coalescence of $\frac{2}{\theta}$ for each pair (recall that the mean of an exponential is 1/rate, so mean time to coalescence for a pair with rate $\frac{2}{\theta}$ is $\frac{\theta}{2}$).

The total coalescence rate when there are k lineages is thus

$$\frac{2}{\theta}\binom{k}{2} = \frac{2}{\theta}\frac{k(k-1)}{2} = \frac{k(k-1)}{\theta}.$$

The density for a coalescent tree is similar to that of a Yule tree. With a coalescing rate of $\frac{2}{\theta}$, the density is

$$f(g|\theta) = \prod_{k=2}^{n} \frac{2}{\theta} \exp(-\frac{k(k-1)}{\theta}(t_{k-1} - t_k)).$$

Note that different time scales may be encountered in the literature: actual calendar time, time scaled so that N generations passes in 1 time unit, time scaled so that $2N$ generations passes in 1 time unit (to account for the fact that in a diploid population of N individuals, there are $2N$ copies of every gene) or evolutionary time where θ is the time unit.

21.6.3 Properties of the coalescent

Let H_n be the height of a coalescent tree with n leaves, then

$$E(H_n) = 2\left(1 - \frac{1}{n}\right).$$

Let L_n be the total length of a coalescent tree with n leaves. Then

$$E(L_n) = 2\sum_{k=1}^{n-1} \frac{1}{k} \approx 2\log(n).$$