Mathematics for Computational Neuroscience & Imaging

John Porrill

Barbie Doll: Math class is tough.
Albert Einstein: Do not worry too much about your difficulties in mathematics,
I can assure you that mine are still greater.

Contents

Part 1

Case Studies

CHAPTER 1

Ordinary Differential Equations

> Science is a differential equation. Religion is a boundary condition. *Alan Turing.*

The theory of differential equations has been one of the most fertile areas in pure mathematics and much of modern mathematics was invented to serve its needs. Our current understanding of the physical world is crucially dependent on differential equations. For example in dynamical systems, that is, systems which change over time, experimental investigation allows us to relate the rate of change to measurable internal and external influences. This process inevitably leads to descriptions of dynamical system behaviour in terms of differential equations.

It is sometimes suggested that this approach, so successful in physics and engineering, is destined to fail when applied to the life sciences. That nothing could be further from the truth is evidenced in neuroscience by the astonishing success of Hodgkin & Huxley's model of neuronal spiking. In fact is reasonable to predict that courses in 'Neuroscience without Calculus' will soon be regarded as just as inadequate a preparation for research in theoretical neuroscience as 'Physics without Calculus' courses are for research in theoretical physics.

We will begin our investigation of differential equations by investigating a first order equation which describes growth and decay processes. It illustrates a surprising range of features characteristic of dynamical systems in general. We will then tackle a second order equation which is capable of describing oscillatory processes. It has important practical applications and great theoretical importance since it describes the small-oscillation modes of general dynamical systems.

1. A First Order System

An important class of growth and decay processes, for example

- population growth processes (e.g. growth of bacterial populations in nutrient media)
- motion of a body in a highly-viscous medium (e.g. rotations of the eye surrounded by orbital tissue)
- charge leakage through an inductor and a resistor in series (e.g. flow of ions through the neural membrane)

are well-described by a very simple differential equation called the *leaky integrator* equation.

Rather than use one of the above physical systems as our concrete concrete example of a leaky integrator we will tackle the problem of filling a leaky bath with water when the plug has been left out. This choice illustrates a wonderful feature of mathematical modelling. Once we have established that a new, and possibly mysterious, system (for example current flow through the neural membrane) is described by the same mathematical model as a simpler system for which our intuitions are more reliable (here the problem of filling a leaky bath) we can explain the properties of the first system by analogy with those of the second.

1.1. Setting up the Mathematical Model. The first and most important stage in mathematical modelling is the translation of the problem into mathematical language. If you get this stage right all your deductive steps from this point on (assuming that you are a good mathematician and work carefully) will at least be very reliable and in principle can be provably true. Any doubt about your conclusions must be justified by pointing out errors in your modelling assumptions. The existence of this bottleneck stage is a major advantage of mathematical modelling. Rational argument is possible if (and in my experience, only if) a acientist makes their assumptions completely explicit in this way.

Unfortunately getting this stage right requires well-established prior knowledge of basic principles supplemented by accurate data. The satisfaction of these two requirements is why physics works so well. Claims by life scientists that it can't work for them are presumably based on the absence of such principles and data in their research area. If so I'm buggered if I know a) why they aren't ashamed, b) why they don't put it right, and c) how they can possibly get around the problem without first feeling (a) then getting on with (b).

Let's start modelling and try to note any questionable assumptions we make. Let $V(t)$ be the volume of water (units litres) in the bath at time t (units seconds). The *state variable* $V(t)$ completely describes the current state of the physical system in a sense to be precisely defined later.

By turning the tap (let's have just one tap) we can adjust the volume of the water flowing into the bath. Call this rate of flow $u(t)$ (units litres per second).

$$\text{flow in} = u(t) \tag{1}$$

Since $u(t)$ can be used to adjust the value of the state variable $V(t)$ it is called a *control variable*.

We need a model for the leakage of water through the plug-hole but the physical process involved here are not as simple as they might seem. A convenient approximation is that the rate of flow is proportional to the pressure difference across the plug hole. If we assume that the bath has a flat bottom (containing the plug) and vertical sides, then this pressure difference is proportional to the depth of water in the bath and the depth of water is itself proportional to the volume of water in the bath.

Putting all that together gives

$$\text{flow out} \propto \text{pressure} \propto \text{depth} \propto \text{volume}$$

and we can express this flow rate as

$$\text{flow out} = kV(t) \tag{2}$$

where the *constant of proportionality* k is a constant scalar parameter. Under normal circumstances this constant will be positive, $k \geq 0$, that is water flows out, not in, through the plug-hole.

> The coefficient k is an example of a *lumped* parameter. It could in priciple be recovered from more fundamental parameters of the physical system such as the area of the bath, the area of the plug hole, the density and viscosity of water, etc. although in practice it would be easier to determine it experimentally as a *phenomenological* parameter, for example by putting a known amount of water in the bath and measuring the volume that escapes through the plug in a given short (why short?) time. It is best to be honest about the difference between lumped/phenomenological parameters and more fundamental parameters. For example channel conductances are in principle open to direct measurement, but this is tedious work, so they are usually measured indirectly using the input-output characteristics of the neuron. This is best thought of as a measurement of a lumped parameter of the input-output system which can be tentatively identified with a fundamental parameter of the physical system.

We can now evaluate the rate of change of volume of water in the bath as

$$\frac{dV}{dt} = \text{flow in} - \text{flow out} = u(t) - kV(t).$$

This equation summarises the information we have about the evolution of a system and is called the equation of motion or *state equation* for the system.

We will often re-order expressions like this to try to emphasise their content. For example we might emphasise the fact that the term $u(t)$ is the *exogeneous input* driving the system by putting it last

$$\frac{dV}{dt} = -kV(t) + u(t) \tag{3}$$

temporarily over-riding the decency rule that minus signs shouldn't be left exposed. Alternatively to emphasise that this is an equation specifying V in terms of u we can put all the unknown terms on the LHS

$$\frac{dV}{dt} + kV(t) = u(t). \tag{4}$$

When presenting a complex equation this kind of notational triviality can help the reader a lot. We will also occasionally denote time derivatives by dots, $\dot{V} + kV(t) = u(t)$, especially for in-line equations.

A system whose state equation has the form above (with $k > 0$) is called a leaky integrator (this name is justified in the next section). This simple equation illustrates a lot more useful terminology:

- the independent variable is time t so the equation describes a *dynamical system*
- it involves a derivative of the state variable V so it is a *differential* equation
- no non-linear terms (such as V^2, $\sin(u)$ or uV) are present so it is a *linear* equation
- there are derivatives with respect to only one variable, time t, so it is an *ordinary* differential equation (or ODE)
- the highest derivative present is the first derivative $\dot{V}$ so it is a *first-order* equation
- the coefficients multiplying the unknown V and $\dot{V}$ (k and 1 respectively) are constant so the equation has *constant coefficients*.

Now we can say a long sentence in fluent maths: the leaky integrator is a dynamical system described by a first-order linear ODE with constant coefficients. If you can reduce a modelling problem to an equation of this type without too much cheating you can feel very pleased with yourself. They have been well-studied and are particularly easy to solve.

1.1.1. *The Exact Integrator.* Now let's look at a special case. Suppose the bath plug is in place, so that $k = 0$, then the rate of change volume is simply the amount of water flowing into the bath. In mathematical terms this becomes

$$\frac{dV}{dt} = u(t). \tag{5}$$

This equation says that $u(t)$ is the derivative of $V(t)$ and so, by the Fundamental Theorem of Calculus, $V(t)$ is given by the integral of $u(t)$

$$V(t) = \int u(t)dt. \tag{6}$$

Hence this very simple dynamical system integrates its input, this is a very useful facility to have available. For example most models of the oculumotor system require at least one integrator in the processing pathway. In the more general case where $k > 0$ the integrator has a leak, hence the name leaky integrator. Leaky

integrators are often used as neural approximations to true integrators since they are much easier to build from biological components.

You may remember from elementary calculus that the *indefinite* integral above (indefinite because no limits of integration are specified) is defined only up to an arbitrary *constant of integration*

$$V(t) = \int u(t)dt + c. \tag{7}$$

Arbitrary constants of integration are characteristic of solutions of differential equations and it is essential to understand why they occur. Suppose we turn on the tap at time 0. Then the amount of water in the bath at time t is the amount V_0 originally in the bath plus the total amount that flows in between the initial time 0 and the final time t (given by a definite integral)

$$V(t) = V_0 + \int_0^t u(\tau)d\tau \tag{8}$$

Clearly the constant of integration c is needed to allow for the different possible values of V_0. Choosing a value for the initial state $V(0) = V_0$ is called a choice of *initial conditions* for the differential equation.

> You might have been tempted to write the equation above as
>
> $$V(t) = V_0 + \int_0^t u(t)dt.$$
>
> but mathematicians frown on the use of t as both the upper limit of an integral and as the dummy variable of integration (specified by dt). To avoid this I have introduced a new integration variable τ (the Greek letter tau; it rhymes with cow). If you wish you can ignore such subtleties but you may find out that the one social skill mathematicians have mastered is the supercilious sneer.

When a solution can be written as a simple formula (what simple means here is not quite well-defined) we often say we have it has an *analytic solution*. Here there is still a (possibly difficult) integral to do so we say that we have an analytic solution *up to a quadrature* (quadrature is the old name for finding the area under a curve).

Analytic solutions to physically important problems are rare but very important when they exist. They often allow us to find simple, informative formulae for quantities of interest. As an example we will tackle the following very simple problem.

> Suppose the tap is turned on fully so that $u(t) = u_{\max}$. How long before the bath is full ($V(t) = V_{\max}$)?

Substituting for $u(t)$ in the equation above gives

$$V(t) = V_0 + \int_0^t u_{\max}d\tau = V_0 + [u_{\max}\tau]_0^t = V_0 + u_{\max}t. \tag{9}$$

Savour this moment: it is a rare event. We have found a *closed form* solution to a differential equation (a closed form expression is one that can be evaluated by a finite, well-defined, sequence of operations).

From this formula we see that the bath will be full ($V = V_{\max}$) when

$$V_0 + u_{\max}t = V_{\max} \tag{10}$$

that is, when

$$t = t_{\text{full}} = \frac{V_{\max} - V_0}{u_{\max}}. \tag{11}$$

Of course this is a trivial problem and the solution should have been obvious: the time required to fill a bath is given by the volume of water required, $V_{\max} - V_0$, divided by the constant flow rate, $u_{\max}$.

Don't be disheartened. The best possible outcome of a modelling study is that after much handle turning the modelling machine pops out an answer you can actually explain to your colleagues! The only drawback is that you will want to publish all the clever maths you used on the way rather than much more trivial argument suggested by your simple result. If you can't fight this temptation, worry not, there are several (widely unread) journals which cater to your special needs.

If you need further justification for working through this problem in such detail then you may like to know that the steps followed here are *exactly* those used to solve the theshold crossing problem for the leaky integrator. In that case they allow you to evaluate a very useful closed-form expression for the firing rate of an integrate-and-fire neuron with constant injected current.

We can also use this problem to work on improving our mathematical personal hygiene. Suppose the solution wasn't so obviously correct. How could you check it without simply re-doing the calculation? A good first step is to perform a *dimensional analysis* to be sure that the result has the right dimensions:

- the numerator (top of fraction) is a volume, measured in litres, the denominator is a flow rate, measured in litres per second, the result is therefore measured in seconds as it should be.

(This step would be easier if neuroscientists, or at least editors of neuroscience journals, recognised that physical quantities do in fact have units, and that to leave them unspecified is a crime against humanity which should make a paper unpublishable). Another useful check is to look for special cases where the answer is obvious:

- if the bath is already full, $V_0 = V_{\max}$, the formula gives $t_{\text{full}} = 0$, so no time is needed to fill a full bath.

If possible check that the solution has the right qualitative properties:

- since $V_{\max} - V_0$ is in the numerator (top of the fraction) bigger baths require longer fill times
- since $u_{\max}$ is in the denominator (bottom of the fraction) higher flow rates lead to shorter fill times .

Always check for features such as negative quantities, divisions by zero, square roots of negative quantities etc., that might lead to physical absurdities:

- when $V_0 > V_{\max}$ the predicted time to fill the bath is negative, but this OK since you are attempting to remove water from a bath that has overflowed via the tap.

Good applied mathematicians, even those who are sloppy calculators, (they can be identified by their catchphrase 'correct up to minus signs and factors of two') don't often make real howlers because they do such reality checks as a matter of course.

1.1.2. *The Transient Solution.* It is a geat temptation to try to solve a problem all at once. The result is likely to be a long, uniformative formula, or, even worse (and a much more likely outcome for the average investigator) a massive, misleadingly precise, computer simulation. Often it is more useful to investigate many special cases to identify behaviours we can expect to find in the general solution.

We have already looked at one special case, the exact integrator with $k = 0$. In that case we simplified the problem by putting the plug in the bath. In this section we will take out the plug but turn off the tap, setting the control input $u(t)$ to zero.

A dynamical system of this kind, which evolves in time with no external inputs, is called without *exogenous* inputs or more concisely an *autonomous* system. Setting the control input to zero in the leaky integrator equation gives

$$\frac{dV}{dt} + kV(t) = 0. \tag{12}$$

The differential equation obtained by setting input terms to zero is often called the *homogeneous* equation. We will look its solution using a range of methods.

1.1.3. *Analytic Solution of the Homogeneous Equation.* This equation can be solved *by inspection* to give

$$V = Ae^{-kt}. \tag{13}$$

This is the kind of statement non-mathematicians hate. It means that you are being expected either a) to know the solution from previous courses or b) to be able guess a solution a solution using your skill and judgement. For example here we need a function whose derivative is very closely related to itself, this points to an exponential function, possibly $V = e^t$, but we need an extra factor of $-k$ in the derivative, so (think chain rule) we make the exponential a function of $-kt$, $V = e^{-kt}$, then we use the fact that any multiple of the solution is also a solution (since the equation is linear in V) to get $V = Ae^{-kt}$. Finally we must check that this solution is correct

$$\frac{dV}{dt} = \frac{dAe^{-kt}}{dt} = \left(\frac{dAe^{x}}{dx} \cdot \frac{dx}{dt}\right)_{x=-kt} = (Ae^{x}) \cdot (-k) = -k(Ae^{-kt}). = -kV \tag{14}$$

With just a little practice all the steps in this equation will simply happen in your head.

The formula above gives a solution of the equation for any value of A. It is in fact the *general solution* of the equation, that is, by varying the arbitrary constant A we get all possible solutions. We can choose the arbitary constant A to satisfy the initial conditions. For example if $V(0) = V_0$ then

$$V_0 = V(0) = Ae^{-k\cdot 0} = Ae^0 = A \cdot 1 = A \tag{15}$$

determines the value of A and so

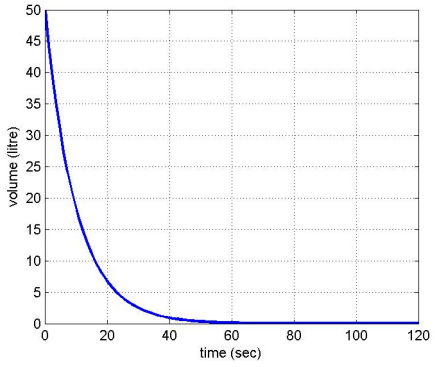
$$V = V_0 e^{-kt}. \tag{16}$$

This example is typical: the general solution of a first-order ODE has just one arbitrary constant which can be fixed by imposing a single initial condition.

At this stage it is important ot either sketch the graph of your solution or let MatLab do it for you:

```
V0 = 50;                    % initial condition
k  = 0.1;                   % rate constant
t = linspace(0, 120, 100); % 2min, 100 timesteps
V = V0*exp(-k*t);
figure; plot(t, V)
```

Figure 1 Plot of analytic solution of the leaky integrator with $V_0 = 0.9$ and $k = 0.1$.



This is the characteristic shape of an *exponential decay* curve. This is what the solution does if it has been prepared in some initial state (specified by the initial conditions) and we leave it to evolve without external infuences. This behaviour is

called the *transient behaviour* of the system. In this example the transient response is clearly *stable*, that is, it decays to zero.

When inspection or inspired guesswork fails to solve your equation there are a numbber of systematic procedures that you can try. Alternatively symbolic mathematics programs like Mathematica and Maple can (in principle) solve all ODE's which have closed form solutions in terms of of a usual set of special functions. It is only fair to admit that ODEs with closed form solutions are rare (in fact there is usually a deep reason why they exist). Hence in genral we have to turn to other methods, both quantitative and qualitative.

1.1.4. *The Characteristic Time.* Some qualitative properties of dynamical systems can be recovered without any attempt to solve the equation of motion. Dimensional analysis is often a good source for such properties. For example let's look at the rate coefficient k. Since $\dot{V}$ can be added to kV these quantities must have the same dimensions, this allows us to calculate the dimensions of k

$$[k] = \frac{[\dot{V}]}{[V]} = \frac{\text{litre sec}^{-1}}{\text{litre}} = \text{sec}^{-1} \tag{17}$$

so k is an inverse time. Its reciprocal

$$T = \frac{1}{k} \tag{18}$$

must have dimensions of time and is therefore a *characteristic time* for the system, called the *time constant* of the leaky integrator.

It is clear from the analytic solution that over the characteristic time T the state variable decays by a factor of about e^{-1}

$$V(T) = V_0 e^{-k\cdot\frac{1}{k}} = e^{-1} = \frac{1}{2.718\ldots} V_0. \tag{19}$$

Somtimes it is more convenient to quote the half-life for a decay process, that is, the time $T_{0.5}$ taken to decay to half the initial value. From $V(T_{0.5}) = \frac{1}{2}V_0$ we deduce that

$$e^{-kT_{0.5}} = \frac{1}{2} \Rightarrow -kT_{0.5} = \log(\frac{1}{2}) \Rightarrow T_{0.5} = -\frac{\log(\frac{1}{2})}{k} = 0.693\ldots T \tag{20}$$

that is, the half life is approximately 70% of the time constant.

1.1.5. *Graphical Solution.* In this section we will try to forget that we have the analytic solution and look for further qualitative insight available in the state equation. The essentially graphical method described below is very powerful and has much deeper implications than appear at first sight.

As we saw in the last section the graph of a solution $V(t)$ of the equation is a curve in the plane with coordinates (t, V). What information about this curve does the state equation contain? Clearly the differential equation tells us that if the curve goes through the point (t, V) it has slope $\dot{V} = -kV$ there.

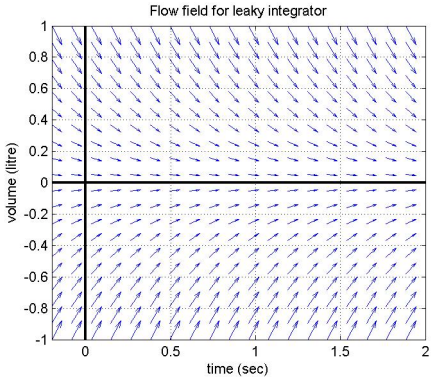
This information can be illustrated graphically by placing a grid on the (t, V) plane and at each grid point drawing an arrow or *flow vector* with slope specified by the differential equation. Such a diagram is called a *flow field* for the differential equation. A rough sketch of the flow field can often be obtained by hand, more accurate plots such as Figure 2 can be obtained using MatLab:

```
t = linspace(-0.2, 2, 20);
V = linspace(-1,   1, 20);
[t, V] = meshgrid(t, V); % 20x20 grid
k = 2;
ft = ones(size(t)); % flow t-cmpnt = 1
fV = -k*V;          % flow V-cmpnt = slope
```

```
figure; quiver(t, V, ft, fV);
```

(in this code each flow vector is given t-component equal to 1 so that its V component is just the required slope $\dot{V} = -kV$. MatLab then scales all the arrows equally so they fit the grid-spacing nicely).

Figure 2 Flow field for the leaky integrator $\dot{U} + \frac{1}{2}U = 0$.



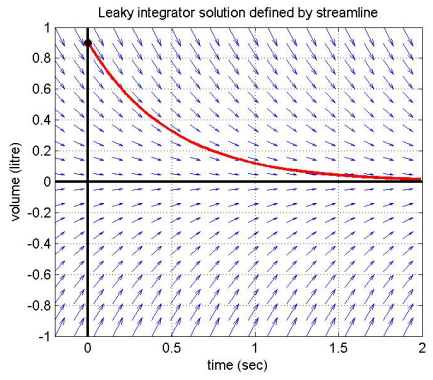
A solution to the differential equation specifies a curve whose slope matches the direction of this flow field at every point. Such a curve is called an *integral curve* of the flow. It is helpful to visualise this flow field as the motion of an imaginary fluid whose velocity at each point is given by the flow vector. An integral curve then corresponds to the path of a small particle moving with the fluid (so that integral curves are often called *stream lines*).

Integral curves can be obtained graphically by drawing a smooth curve everywhere tangential to the flow field. Obviously there is a continuous family of such integral curves. A unique curve can be specified by choosing a starting point (specified by the initial conditions) and each streamline corresponds to a different choice of initial conditions. Specifying the initial condition $U(0) = U_0$ is equivalent to specifying the starting point$(0, U_0)$ on the $t = 0$ axis for the stream line.

In Figure 3 a stream line is overlayed on the flow field of Figure 2 using the MatLab utility `streamline`

```
t0 = 0; V0 = 0.9; % starting point
streamline(t, V, ft, fV, t0, V0);
```

Figure 3 Streamline with initial conditions $U_0 = 0.9$ superposed on flow field from Figure 2



The flow field discussed here is slightly more general than the *phase portrait* to be defined later. In fact it is the phase portrait for the augmented system

$$\frac{dV}{d\tau} = -kV(\tau) \tag{21}$$

$$\frac{dt}{d\tau} = 1 \tag{22}$$

in which time is added as an extra dependent variable. This is a trivial but surprisingly powerful trick which turns up repeatedly in control theory.

1.1.6. *Numerical Solution.* The flow field picture defined above is a nice way to visualise how numerical solution methods work. If we choose a starting point then the flow field at that point tells us the initial direction of the streamline. Moving a *small* distance in this direction gives a new point on the streamline; we can compute the flow vector at this new point to make a further step alog the streamline. Iterating this procedure allows us to plot the whole streamline. Clearly this is an approximation since the method approximates the actual stream line by a series of short line segments but for small enough steps it will be a good approximation.

The Euler method for solving first order ODEs implements this procedure. First we must choose a small time step Δt. We will attempt to evaluate $V(t)$ only at the times $t = 0, \Delta t, 2\Delta t, 3\Delta t, \ldots$ that is at

$$t_i = i\Delta t \qquad i = 0, 1, 2, \ldots \tag{23}$$

to give the *discrete* approximation

$$V_i \approx V(t_i) \qquad i = 0, 1, 2, \ldots \tag{24}$$

Approximating the streamline by a tangential straight line with the same slope is equivalent to using the first order Taylor approximation

$$V(t + \Delta t) \approx V(t) + \frac{dV}{dt}\Delta t = V(t) - kV(t)\Delta t \tag{25}$$

where the derivative is specified by the differential equation $\dot{V} = -kV$. Applying this rule iteratively starting at $t = 0$ gives a *recurrence relation*

$$V_{i+1} = V_i - kV_i\Delta t \tag{26}$$

which allows us to write very simple code fo obtain an approximate solution with given initial conditions

```
k = 0.5;
nt = 200; dt = 0.01; t = (0:nt-1)*dt;
V(1) = 0.9;
for i = 1:nt
    V(i+1) = V(i)-k*V(i)*dt;
end
figure; clf; plot(t, V)
```

to calculate and plot the Euler approximation to the solution with initial conditions $V_0 = 0.9$.

MatLab starts indexing its arrays at $i = 1$ while some other languages start at $i = 0$ or allow arbitrary starting points. It seem MatLab policy is to use $i = 1$ conventions consistently in both documenting mathematical formula and code. I prefer to use the (usually neater) $i = 0$ convention for maths, but then have to convert to $i = 1$ conventions when writing MatLab code. This causes its fair share of programming errors. *Note to MatLab: implement variable starting index arrays.*

The *update rule* above can be written as

$$V_{i+1} = (1 - k\Delta t)V_i.$$

which is a simple *recurrence equation* and can be solved by inspection:

$$V_i = (1 - k\Delta t)^i V_0.$$

The Euler method gives the above geometric series rather than the exact exponential

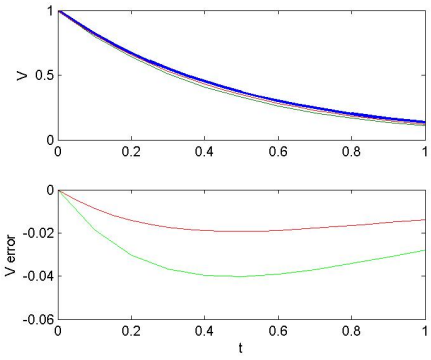
$$V_i = e^{-ik\Delta t} V_0 = (e^{-k\Delta t})^i V_0 \tag{27}$$

so the decay constant at each step has been approximated as

$$e^{-k\Delta t} \approx 1 - k\Delta t. \tag{28}$$

whih is the first order Taylor approximation to the exponential function ($e^x \approx 1 + x + \frac{x^2}{2} + \ldots$). Since the errors ignored are quadratic or higher in Δt, halving the time step will reduce the approximation error at each individual step by a factor of about 4. Because we now require twice as many steps to reach any given time the approximation error at a given time is only reduced by a factor of 2. This so-called *linear* convergence behaviour is illustrated in Figure 4. Linear convergence is not very good and much better approximation schemes have been devised. Some of the best are implemented in the black-box differential equation solvers supplied with MatLab.

Figure 4 Solving $\dot{V} + 2V = 0$ using the Euler method. Top plot: The thick blue curve is the true exponential solution, the green and red curves are the approximations using 10 and 20 time step respectively. Lower plot: Errors in these two approximations. It can be seen that doubling the number of steps (equivalently halving the discretisation time) approximately halves the error in the Euler method.



It is important to note that the Euler scheme also has the potential for catastrophic divergence. Despite these drawbacks it is a very useful approximation scheme for many purposes.

1.2. Using Black-Box ODE Routines. By using better approximations than line segments, for example segments of polynomial curves, and more intelligent control architectures, algorithms with higher order convergence and improved stability can be developed. These higher order schemes are usually accessed via so-called black-box routines. Learning how to use these gives us a tool that can be used to solve equations of almost arbitrary complexity.

Black-box routines solve ODEs of the general form

$$\frac{dy}{dt} = f(t, y(t)) \tag{29}$$

where the RHS can be an arbitrary function of time t and the state variable y.

In MatLab the user is required to provide code to evaluate the function on the RHS which must be specified in the standard form

```
function ydot = func(t, y)
% code evaluating ydot
% goes here e.g.
ydot = -y^2+sin(y)^3+t*y
```

The equation can be solved over a given time interval using the black-box routine `ode23`

```
[t, y] = ode23(@func, tspan, y0);
```

where the user passes the pointer `@func` to the function `func`, the time range over which to solve the equation (as a 2-vector `tspan=[t0 t1]`) and the initial value `y0` of the state variable. The routine returns a vector of times `t` in the range `tspan` and a vector of associated function values `y` at those times.

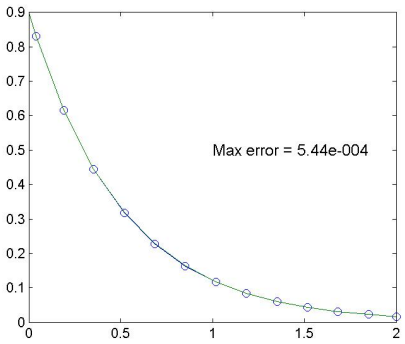
Using these procedure we can solve the leaky integrator with given initial conditions and plot the solution (see Figure 5 using the code below

```
function plot_lky
V0 = 0.9;
tspan = [0 2];
[t, V] = ode23(@lky, tspan, V0)
figure; plot(t, V)

function Vdot = lky(t, V)
Vdot = -2*V;
```

(making this a function rather than a script allows us to put it all in one file).

Figure 5 Solving $\dot{V} + 2V = 0$ using the black-box method ode23. Blue curve is the true solution, green is the approximation. The maximum error is about 0.0005. The error for the Euler method with the same number of function evaluations is about 30 times larger.



It is very important not to get too blasé about the use of black-box numerical routines. They are often slow, over-complex and bug-ridden. They usually have many default parameters set that must be adjusted by the user for optimum performance. It's also important to know how they work: for example a predictor-corrector method will give meaningless results or be hopelessly inefficient for noisy inputs.

CHAPTER 2

Linear Algebra

Neo: What is the Matrix?
Trinity: The answer is out there, Neo, and it's looking for you, and it will find you if you want it to. *The Matrix (1999)*

1. Example: An Idealised Linear Neuron

A natural context to discuss the usefulness of linear algebra in neuroscience is the linear neuron (this has a lot of diferent names depending on the application area). It is a very simple one layer neural net with multiple inputs and just one output.

To make the model concrete let's imagine a neuron $\mathcal{N}$ taking input from n sensory neurons. We will assume that the relevant measure of the input at the j-th synapse on $\mathcal{N}$ is the firing rate x_j of the pre-synaptic neuron.

I said we should be careful about units. Firing rates are usually measured as number of spikes per second so that x_j has dimensions sec^{-1} (some people, including me, refer to firing rates in units of Hz (Hertz) but that's not strictly correct because Hz means *cycles* per second).

The input to the neuron is assumed to be completely described by the instantaneous firing rates $x_1, x_2, \ldots, x_n$ of the input neurons. We also assume that the output of the neuron is adequately decribed by its firing rate y. The linear neuron, unlike the leaky integrator, has no internal state to remember the history of its inputs (real neurons are not quite like this). Because of this simplification the output firing rate y can only depend on the instantaneous inputs x_i.

Clearly the firing rate is a positive quantity. However it is often firing modulation, that is, changes relative to the tonic/spontaneous firing rate of the neuron that carries information. In this section we will adopt the convention that a value $y = 0$ means that the linear neuron is firing at its tonic rate, while positive and negative y indicate true firing rates above and below the tonic rate respectively. Hence all the quantities x_j, y can be either positive or negative.

> It is of great importance to experimental neuroscientists to establish whether neuronal outputs are excitatory or inhibitory. This distinction is chemical rather than computational and loses much of its importance in systems neuroscience. An inhibitory neuron can easily have an excitatory effect (relative to the status quo) on down stream neurons if its firing rate decreases below its tonic level.

We assume that this dependence of y on the x_i is linear, so that the linear neuron calculates a weighted sum of its inputs

$$y = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n = \sum w_j x_j. \tag{30}$$

The coefficient w_j is called the *weight* or *efficacy* of the j-th synapse. Since synapses can be excitatory or inhibitory weights can be either positive or negative.

Dimensional consistency requires that

$$[y] = [w_j][x_j] \tag{31}$$

and since y and x_j both have the same dimensions ($\sec^{-1}$) the weights w_j must be dimensionless.

This may seem to be very unlike any real neuron, and often this difference is emphasised by neuroscientists who will say things like "property X - which I have just discovered - gives neuron Y the computing power of a Pentium chip". The commonly assume that linear devices are too trivial for them to spend time on. In fact from a theoretical point of view linearity is the most important property and unlikely a mathematical object can have, and in practical applications such as electronics linear components are difficult to make and absolutely essential to all kinds of devices. This is presumably why biology often goes to extraordinary lengths to linearise various subsystems, and one would expect individual neurons to be no exception. A neuron would be expected to have at least a small linear operating regime because of the following argument: small changes in input firing rates about some tonic level $x_i^{(0)}$ will produce small changes in output firing rate around some tonic level $y_{(0)}$. A Taylor approximation to first order predicts

$$y - y^{(0)} \approx \sum \frac{\partial y}{\partial x_j}(x_j - x_j^{(0)})$$

so we expect small fluctations about the tonic levels to satisfy the linear neuron equation

$$\delta y \approx \sum w_i \delta x_i \qquad \text{with} \qquad w_j = \frac{\partial y}{\partial x_j}.$$

It is up to the 'non-linearist' to demonstrate that this potentially extremely useful linear regime is not the usual operating range of the neuron.

1.1. Vectors. It is convenient to think of a list of n scalars such as the inputs x_j to the linear neuron as a single object

$$\mathbf{x} = (x_1, x_2, \ldots, x_n) \tag{32}$$

called an *n-vector* or simply a vector. When written out as a row of numbers as above the vector is called a row vector. In fact it is more conventional to write vectors as columns

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \tag{33}$$

(this is not just a notational point, more about this below).

In MatLab vectors are a basic data structure. A row or column 3 vector `x` with components $1, 2, 3$ can be constructed as follows

```
>> xrow = [10 20 30]
     ans = 10 20 30
>> xcol = [4; 5; 6]
     ans = 4
           5
           6
```

and its i-th component can be accessed as

```
>> xrow(2) % value of 2nd entry
        ans = 20
>> i = 3;
>> xcol(i) % value of i-th entry
        ans = 6
```

1.2. Arithmetic Operations. There are two basic arithmetic operations defined on vectors. We will write these out in detail for column vectors only.

Firstly a vector $\mathbf{x}$ can multiplied by a scalar λ (greek letter lambda) by multiplying all its components by λ

$$\lambda\mathbf{x} = \lambda \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \equiv \begin{pmatrix} \lambda x_1 \\ \lambda x_2 \\ \vdots \\ \lambda x_n \end{pmatrix}. \tag{34}$$

In MatLab this operation is denoted by `lambda*x` as in

```
>> lambda = 2;     % no Greek letters in MatLab!
>> x = [1; 2; 3];
>> lambda*x % query value
        ans = 2
              4
              6
```

Secondly two vectors $\mathbf{x}$ and $\mathbf{y}$ can be added by adding corresponding components

$$\mathbf{x} + \mathbf{y} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \equiv \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{pmatrix}. \tag{35}$$

In MatLab this operation is denoted by `x+y` as in

```
>> x = [1; 2; 3];
>> y = [3; 2; 1];
>> x+y % query value
        ans = 4
              4
              4
```

Rather than write out all the components of a vector equation it is sometimes useful to use a notation like

$$\mathbf{x} - \mathbf{y} = (x_i) - (y_i) \equiv (x_i - y_i) \tag{36}$$

(this equation defines vector subtraction).

The zero vector has all its its components zero. It is usually denoted by the ame symbol, 0, as the scalar zero so it has to be recognised by its context, as in $\mathbf{x} - \mathbf{x} = 0$.

1.3. Vectors in n-Dimensional Space. The components of a 2-vector (x_1, x_2) can be thought of as the Cartesian coordinates of a point in a 2-dimensional space. Similarly components of a 3-vector (x_1, x_2, x_3) can be thought of as the Cartesian coordinates of a point in a 3-dimensional space.

It is natural to generalise this and think of an n-vector as the coordinates of a point in an n-dimensional space. For $n > 3$ (or $n > 4$ if you can imagine time as the fourth dimension) these spaces are not easy to imagine, but it turns out that drawing 2D and 3D diagrams can often supply useful intuitions about problems in higher dimensional spaces. The n-dimensional space with real numbers as coordinates is usually denoted by $\mathbb{R}^n$, so that $\mathbb{R} = \mathbb{R}^1$ is the real line, $\mathbb{R}^2$ is the real plane and $\mathbb{R}^3$ describes 3D space.

1.3.1. *Length of a Vector.* Many geometrical notions transfer directly into higher dimensions. For example we can assign a length $|\mathbf{x}|$ to a vector $\mathbf{x}$. In 2 dimensions the length of a vector (x_1, x_2) is given by Pythagoras' formula

$$|\mathbf{x}| = \sqrt{x_1^2 + x_2^2} \tag{37}$$

the in higher dimensions this formula can be generalised to give the *definition* of length

$$|\mathbf{x}| \equiv \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2} = \sqrt{\sum x_j^2} \tag{38}$$

(this quantity is also called the *magnitude* or *norm* of the vector). In MatLab it can be calculated using the function `norm`

```
>> u = [3 4 0];
>> norm(u)
   ans = 5
```

Clearly the zero vector has zero length. If a vector has length 1

$$|\mathbf{x}| = 1 \tag{39}$$

it is called a unit vector. Unit vectors are sometimes distinguished by a hat $\hat{\mathbf{x}}$. A non-zero vector can be *normalised*, that is, made into a unit vector, by dividing all its components by its own length

$$\hat{\mathbf{x}} = \frac{1}{|\mathbf{x}|}\mathbf{x} \tag{40}$$

e.g. in MatLab

```
>> u = [3 0 4];
>> uhat = u/norm(u)
   uhat = [0.6 0 0.8]
>> norm(uhat)
   ans = 1
```

1.3.2. *The Dot Product.* A geometrical construct that may not be so familiar as length is the the *dot* (or *scalar*, or *inner*) product of two vectors. For n-vectors $\mathbf{u}$ and $\mathbf{v}$ this is defined as

$$\mathbf{u} \cdot \mathbf{v} = u_1 v_1 + \ldots + u_n v_n = \sum u_i v_i. \tag{41}$$

In MatLab this expression can be expressed very compactly, for example

```
>> u = [1 2 3];
>> v = [1 1 0];
>> d = sum(u.*v) % multiply elements and sum
   ans = 3
```

The dot product of a vector with itself is clearly the square of its length

$$\mathbf{u} \cdot \mathbf{u} = \sum u_i^2 = |\mathbf{u}|^2 \qquad |\mathbf{u}| = \sqrt{\mathbf{u} \cdot \mathbf{u}} \tag{42}$$

In 2 and 3 dimensions the dot product is related to the angle θ between the two vectors and their lengths by the cosine rule

$$\mathbf{u} \cdot \mathbf{v} = |\mathbf{u}||\mathbf{v}| \cos\theta \tag{43}$$

In higher dimensions this relationship can be used to *define* the notion of angle $\theta = \angle(\mathbf{u}, \mathbf{v})$ between two vectors using the formula

$$\cos\theta = \frac{\mathbf{u} \cdot \mathbf{v}}{|\mathbf{u}||\mathbf{v}|} \tag{44}$$

A very important situation is when the angle between two vectors is 90° ($\pi/2$ in grown up units of radians). Since $\cos 90^\circ = 0$ this happens when the dot product is zero

$$\mathbf{u} \cdot \mathbf{v} = 0. \tag{45}$$

In this case the vectors are said to be perpendicular or more commonly *orthogonal* and the above equation is called the orthogonality condition. The zero vecor is trivially orthogonal to all vectors.

If two (non-zero) vectors are multiples of each other

$$\mathbf{u} = \lambda \mathbf{v} \tag{46}$$

then

$$\cos\theta = \frac{\mathbf{u} \cdot \mathbf{v}}{|\mathbf{u}||\mathbf{v}|} = \frac{\lambda \mathbf{u} \cdot \mathbf{u}}{\lambda |\mathbf{u}||\mathbf{u}|} = 1 \tag{47}$$

so that the angle between them is $\theta = \cos^{-1} 1 = 0$. Since the angle between them is zero we say that they lie in the same direction. For example the normalisation procedure above allows us to construct a unit vector $\hat{\mathbf{x}}$ in the same direction as any vector $\mathbf{x}$.

1.4. Geometry of the Linear Neuron. If we describe the neuronal inputs and synaptic weights by vectors $\mathbf{w}$ and $\mathbf{x}$ then the output of the linear neuron $y = \sum w_j x_j$ can be written very compactly using the dot product as

$$y = \mathbf{w} \cdot \mathbf{x}. \tag{48}$$

The output of the linear neuron is zero precisely when the input vector $\mathbf{x}$ is orthogonal to the weight vector $\mathbf{w}$. We shall see later that the set of all inputs which give zero output forms a *hyperplane* in input space, called the null space of $\mathbf{w}$.

It is clear that we can get very large outputs from a neuron by haveing some inputs or weights which are very large. Suppose we normalise out this dependence by restricting ourselves to unit input and weight vectors $\hat{\mathbf{x}}, \hat{\mathbf{w}}$. Then the firing rate is just the cosine of the angle between these two vectors

$$y = \hat{\mathbf{w}} \cdot \hat{\mathbf{x}} = \cos\theta. \tag{49}$$

This output is zero, as noted above, when the vectors are orthogonal. It takes its maximum value of 1 when the angle between the vectors is 0°. That is, the linear neuron is optimally tuned to detect input vectors lying in the same direction as the weight vector.

Hence a linear neuron can be thought of as a detector which is maximally sensitive to inputs lying along a single direction in input space, and minimally sensitive to inputs lying in the $n-1$-dimensional hyperplane orthogonal to this direction.

2. Matrices

Now suppose we have m neurons all taking the same inputs x_j and that the output of the i-th neuron is y_i. The axon carrying signal x_j now synapses on m different neurons so we need a notation for its synaptic weight on the i-th neuron; we will call this weight w_{ij}.

The relationship between the output of this single layer neural net and its inputs is completey described by the doubly-indexed array of scalars w_{ij}. The output of the i-th neuron is

$$y_i = w_{i1}x_{i1} + w_{i2}x_{i2} + \ldots + w_{in}x_{in} = \sum w_{ij}x_j. \tag{50}$$

It is convenient to think of a doubly indexed array of $m \times n$ scalars such as the weights w_{ij} above as a single object W called an $m \times n$ matrix

$$W = (w_{ij}) = \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \dots & \dots & \dots & \dots \\ w_{m1} & w_{m2} & \dots & w_{mn} \end{pmatrix} \tag{51}$$

Conventionally the first index indicates row position and the second index column position. In MatLab ("Matrix Laboratory") matrices are the basic data structure. A column 2×3 matrix with components

$$W = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

can be constructed and accessed as follows

```
>> W = [1 2 3; 4 5 6]
        ans = 1 2 3
              4 5 6
>> W(2, 3)
        ans = 66
```

Rows and columns can be extracted from using the colon operator

```
>> W(1, :) % first row
        ans = 1 2 3
>> W(:, 3) % third column
        ans = 3
              6
```

Matrices with only one row or column are called row or column matrices as well as row and column vectors.

2.1. Matrix Transposition. Although it usual to write vectors as column vectors this is only a convention. MatLab functions with vector output usually produce row vectors by default e.g.

```
>> x = 1:4
        ans = 1 2 3 4
>> x = linspace(0, 1, 5)
        ans = 0 0.25 0.5 0.75 1
```

A useful operation in this context is Matrix transposition, which swaps the rows and columns of a matrix. Transposition is usually denoted by a dash or superscript T so that

$$V = W^T \Leftrightarrow V_{ij} = W_{ji} \tag{52}$$

for example

$$W = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \qquad \text{has transpose} \qquad W^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}.$$

In MatLab transposition is notated by a dash

```
>> W = [1 2 3; 4 5 6]
        W = 1 2 3
            4 5 6
>> W'
        ans = 1 4
```

```
2 5
3 6
```

Clearly the transpose of a row vector is a column vector and vice versa

$$\begin{pmatrix}1\\2\\3\end{pmatrix}^T = \begin{pmatrix}1 & 2 & 3\end{pmatrix}$$

and this is sometimes used to format column vectors more concisely, as in

$$\mathbf{x} = \begin{pmatrix}x_1 & x_2 & \dots x_N\end{pmatrix}^T.$$

We are being a bit cavalier with concepts here. Is a vector really a column matrix? Are row vectors different from column vectors? More on this later.

2.2. Matrix Arithmetic Operations. As with vectors, the basic arithmetic operations can be defined on matrices. The first is multiplication by a scalar

$$\lambda W = \lambda(w_{ij}) = (\lambda w_{ij}) \tag{53}$$

this is simply elementwise multiplication. In MatLab this would be

```
>> lambda = 2;
>> W = [1 2 3; 4 5 6];
>> lambda*W
        ans = 2  4  6
              8 10 12
```

The second is addition of two matrices V and W of the same size

$$V + W = (v_{ij}) + (w_{ij}) = (v_{ij} + w_{ij}) \tag{54}$$

which is simply elementwise addition.

```
>> V = [0 0; 0 1];
>> W = [1 2; 3 4];
>> V+W
        ans = 1 2
              3 5
```

2.3. Matrix Multiplication. What makes matrices really interesting is a third operation, called matrix multiplication. It might seem that a natural matrix product would take two matrices of the same size and multiply them elementwise

$$V \odot W = (v_{ij}) \odot (w_{ij}) = (v_{ij} w_{ij}). \tag{55}$$

This operation (sometimes called the Schur product) is useful in some contexts (for example image processing). In MatLab it is implemented as the elementwise product `V.*W`, for example

```
>> V = [1 1; 2 2];
>> W = [1 2; 3 4];
>> V.*W
        ans = 1 2
              6 8
```

The elementwise product is a poor relation to the much more interesting matrix product. We will build this operation up in stages.

2.3.1. *Product of a Matrix and a Vector.* We start with the equation above for the output of a linear neuron

$$y_i = \sum_{j=1}^{m} w_{ij} x_j. \tag{56}$$

and use this to define the product $W\mathbf{x}$ of the $m \times n$ matrix $W = (w_{ij})$ and a *column* n-vector $\mathbf{x} = (x_j)$

$$\mathbf{y} = W\mathbf{x} \Leftrightarrow y_i = \sum_{k=1}^{m} w_{ij} x_j. \tag{57}$$

Now we can write the equation for the outputs of the neuronal assemby in the compact form

$$\mathbf{y} = W\mathbf{x}. \tag{58}$$

Notice that this defines a mapping (usually also called W)

$$W : \mathbb{R}^n \to \mathbb{R}^m \qquad W : \mathbf{x} \to \mathbf{y} = W\mathbf{x}. \tag{59}$$

This transformation "squashes and squeezes" vectors input space to give vectors in output space. Getting an feel for the nature of these *linear transformations* is a really useful mathematical tool.

In MatLab the matrix-vector product is simply denoted by $*$

```
>> W = [1 1; 2 2];
>> x = [2; 3];
>> W*x
        ans = 5
              10
```

It is worth learning how to do this calculation by hand.

2.3.2. *Product of two Matrices.* I will derive the matrix product in a concrete application. Suppose we put two single layer nets end-to-end to form a 2-layer net. The m firing rates $\mathbf{y}$ output by the first layer are used as inputs to a second layer of p neurons whose outputs form the p-vector $\mathbf{z}$. The synaptic weights for this second layer form a $p \times n$ matrix V. The computations performed by these two layers are

$$\mathbf{y} = W\mathbf{x} \qquad \mathbf{z} = V\mathbf{y}. \tag{60}$$

so we have an overall transformation

$$\mathbf{x} \xrightarrow{W} \mathbf{y} \xrightarrow{V} \mathbf{z} \tag{61}$$

$$\mathbb{R}^n \xrightarrow{W} \mathbb{R}^m \xrightarrow{V} \mathbb{R}^p. \tag{62}$$

To characterise the overall transformation of inputs $\mathbf{x}$ into outputs $\mathbf{z}$ will involves a short calculation. From the definition of the matrix-vector product we have

$$z_i = \sum_j V_{ij} y_j = \sum_j V_{ij} \left(\sum_k W_{jk} x_k) \right) = \sum_k \left(\sum_j V_{ij} W_{jk} \right) x_k \tag{63}$$

and the last term has exactly the form of a matrix-vector multiplication

$$z_i = \sum_k P_{ik} x_k \tag{64}$$

where the new matrix P has elements

$$P_{ik} = \sum_j V_{ij} W_{jk}. \tag{65}$$

We call this matrix the matrix product $P = VW$.

Although the multiplication rule may look complex the operation it describes is very common and natural. The matrix W transforms n-vectors $\mathbf{x}$ into m-vectors $\mathbf{y}$. The matrix V transforms m-vectors $\mathbf{y}$ into p-vectors $\mathbf{z}$. The product matrix $P = VW$ defined above represents the overall transformation from m-vectors $\mathbf{x}$ into p-vectors $\mathbf{z}$

$$\mathbf{z} = V\mathbf{y} = V(W\mathbf{x}) = (VW)\mathbf{x}. \tag{66}$$

In MatLab this product is denoted by `V*W`. The following code segment checks that MatLab's implementation of the matrix product is correct

```
>> V = [1 0; 1 1];       % 2x2 matrix
>> W = [1 2 3; 1 1 1];   % 2x3 matrix
>> x = [0; 1; 0];        % 3-vector
>> y = W*x    % x->y
        ans = 2
              1
>> z = V*y    %y->z
        ans = 2
              3
>> P = V*W    % matrix product
        ans = 1 2 3
              2 3 4
>> P*x        % check x->z; OK!
        ans = 2
              3
```

> The consequence for the two-layer net is quite interesting and profound. It means that for linear neurons two-layer nets can always be replaced by a one-layer net with synaptic weights defined by the matrix product formula. This is sometimes misunderstood to suggest that linear neurons aren't very useful.

The *unit* or *identity* matrix is the $n \times n$ matrix

$$I = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 \end{pmatrix} \tag{67}$$

(sometimes called I_n) which has 1's on the diagonal and zeros elsewhere. It has the important property of leaving all n-vectors unchanged under multiplication

$$I\mathbf{x} = \mathbf{x}. \tag{68}$$

Hence its action as a transformation is just to leave its the inputs unchanged. The neural net it describes produces outputs equal to its inputs. In the brain this useful function would be better implemented by a fibre tract rather than a neural net (though some procesing stages in the brain mysteriously seem to do little more than this).

If the dimensions are compatible multiplication by a unit matrix on either left or right leaves the matrix unchanged, that is

$$AI = A \qquad IB = B \tag{69}$$

for all $m \times n$ matrices A and $n \times m$ matrices B. It therefore behaves like a matrix generalisation of the unit scalar 1. In MatLab the $n \times n$ identity matrix is constructed using the function `eye` e.g.

```
>> I = eye(3)
        I =     1  0  0
                0  1  0
                0  0  1
```

3. Matrix Inverses and Solving Linear Equations

A matrix W describes a linear transformation

$$W : \mathbf{x} \to \mathbf{y}. \tag{70}$$

A very basic questions in mathematics is whether a given transformation can be reversed, and, if so, how. For our single-layer neural net the corresponding question is: given an output $\mathbf{y}$ can we find the input $\mathbf{x}$ that produced it? If we can do this for any $\mathbf{y}$, and the corresponding $\mathbf{x}$ is unique, then the transformation is said to be invertible. That is, we can find a mapping

$$V : \mathbf{y} \to \mathbf{x} \tag{71}$$

such that for all $\mathbf{x}$ in $\mathbb{R}^n$

$$\mathbf{x} \xrightarrow{W} \mathbf{y} \xrightarrow{V} \mathbf{x} \tag{72}$$

This problem can be thought of as attempting to determine n independent quantities x_j given the m measurements y_i, and it seems reasonable to begin by looking at the case $m = n$, so that the dimensions of the input and output spaces are the same. In that case W is a square matrix.

We are clearly asking whether multiplication of a vector by a square matrix can be reversed. Consider the operation 'multiply by a scalar k'. This can be reversed by performing the operation 'multiply by the inverse of k, that is, $k^{-1} = 1/k$' (the only exception being when $k = 0$), that is

$$k^{-1}(kx) = x \tag{73}$$

for all scalars x.

The transformation 'multiply by a matrix W' is reversible if there is a matrix $V = W^{-1}$ such that

$$W^{-1}(W\mathbf{x}) = \mathbf{x} \tag{74}$$

for all vectors $\mathbf{x}$. If such a matrix exists it is called the inverse of W. When an inverse exists it is both a left and right inverse the property

$$W^{-1}W = WW^{-1} = I. \tag{75}$$

Calculating the inverse of a large matrix is no easy matter. MatLab finds inverses efficiently and pretty much as accurately as possible using the function `inv`

```
>> A = [1 2; 0 2]
>> Ainv = inv(A)
        Ainv = 1.0000   -1.0000
                    0    0.5000
>> A*Ainv
        ans =     1     0
                  0     1
```

Some square matrices have no inverse and a non-invertible matrix is said to be *singular* (this is like the exception $k = 0$ for the scalar case). If an inverse does not exist it is for a very good reason. It happens when there is a non-zero vector $\mathbf{x}$ such that $W\mathbf{x} = 0$; $\mathbf{x}$ is then called a null-vector of W. If $\mathbf{x}$ is a null-vector then there

are two different vectors, 0 and $\mathbf{x}$, which map to 0 under W, hence W cannot have an inverse (in general a function must be *one-to-one* to have an inverse).

Another test for singularity of a matrix W is to calculate its *determinant*, a scalar quantity denoted by $\det(W)$ (sometimes $|W|$) which we will discuss later. A matrix is singular if

$$\det(W) = 0 \tag{76}$$

For example in MatLab

```
>> W = [1 1 1; 1 2 3; 2 3 4]
      W = 1 1 1
          1 2 3
          2 3 4
>> det(W)
      ans = 0
>> inv(W)
     Warning: Matrix is singular to working precision.
     ans =
          Inf   Inf   Inf
          Inf   Inf   Inf
          Inf   Inf   Inf
```

In elementary algebra the operation we are describing here is called solving a system of linear equations. In this case the system is

$$\begin{aligned} w_{11}x_1 + w_{12}x_2 + \ldots + w_{1n}x_n &= y_1 \\ w_{21}x_1 + w_{22}x_2 + \ldots + w_{2n}x_n &= y_2 \\ &\ldots \\ w_{n1}x_1 + w_{22}x_2 + \ldots + w_{nn}x_n &= y_n \end{aligned}$$

You may have learned how to solve such equations by elimination (many years ago you would have learnt Cramer's rule). The solution proposed here is to write this as a matrix equation $W\mathbf{x} = \mathbf{y}$ and solve it using the matrix inverse as $\mathbf{x} = W^{-1}\mathbf{y}$. This is easily implemented in MatLab. For example to solve

$$\begin{aligned} 2x + 3y + z &= 6 \\ x - y + z &= 1 \\ x + y + z &= 3 \end{aligned}$$

we use the code

```
>> W = [2 3 1; 1 -1 1; 1 1 1];
>> Y = [6 1 3];    % W*X = Y
>> X = inv(W)*Y
     ans = 1
           1
           1
```

so the solution is $x = y = z = 1$.

> As noted above calculating inverses of large matrices is expensive and prone to error. It turns out that there are better algorithms for solving large systems of linear equations than simply calculating an inverse matrix. To use one of these in MatLab replace `X = inv(W)*Y` by the *right-division* operation `X = W\Y`.

4. Least Squares Approximation

"I try (information)
Yes I try (too much information)
Why should I try (information)
Cause I try (too much information)"
Too Much Information, *Duran Duran*

4.1. Additive Noise. Let's consider a situation where the outputs y_i of the output are subject to errors e_i due to neural noise

$$\mathbf{y} = W\mathbf{x} + \mathbf{e} \tag{77}$$

(this noise model is called *additive noise*). We will assume that $m > n$ so there a more (perhaps many more) equations than unknowns. In this situation we are trying to determine the input given a lot of noisy output information. How can we get a good estimate of the input vector in this situation?

> This problem is a very simple example of population coding. The information about a small number of inputs $\mathbf{x}$ is coded by the noisy outputs $\mathbf{y}$ of a large population of linear neurons. We wish to decode this population output.

Suppose we choose a value for the input $\mathbf{x}$. then the associated errors due to noise would have to be

$$\mathbf{e} = \mathbf{y} - W\mathbf{x}. \tag{78}$$

If we know that the true noise levels are small it is reasonable to choose a value $\mathbf{x}$ which makes $\mathbf{e}$ as small as possible. A good measure of how small $\mathbf{e}$ is, is the squared length

$$E(\mathbf{x}) = \frac{1}{2}|\mathbf{e}|^2 = \frac{1}{2}\sum e_i^2 \tag{79}$$

which is also the sum-square-error (ignore the factor half for now). I have written E as a function of $\mathbf{x}$ because the sum-square error depends on our choice of $\mathbf{x}$.

The least-squares solution makes this quantity as small as possible; this is written

$$\hat{\mathbf{x}} = \operatorname*{argmin}_{\mathbf{x}} E(\mathbf{x}) \tag{80}$$

(the hat is conventionally used here to denote an estimate, and $\operatorname{argmin}_{\mathbf{x}}$ means "the argument $\mathbf{x}$ which minimises").

We will be doing a lot more on least squares estimation. Here I just want to write down the solution with minimal justification. It turns out that the least squares estimate is given by the matrix formula

$$\hat{\mathbf{x}} = (W^TW)^{-1}W^T\mathbf{y}. \tag{81}$$

We should check, firstly, that this formula makes sense. Since W is $m \times n$ and W^T is $n \times m$ we *can* multiply them (since the inner dimensions match). The result W^TW is square (its size is $m \times m$) so if it is non-singular (assume this for the moment) it *can* be inverted to get $(W^TW)^{-1}$. Secondly we should check what happens if there is no noise If $\mathbf{y} = W\mathbf{x}^*$ then

$$\hat{\mathbf{x}} = (W^TW)^{-1}W^T\mathbf{y} = (W^TW)^{-1}W^T(W\mathbf{x}^*) = (W^TW)^{-1}(W^TW)\mathbf{x}^* = \mathbf{x}^*, \tag{82}$$

the estimate gives the true value just as it should.

The least squares formula above can be implemented directly in MatLab

```
>> W = randn(100, 3);      % a big, random, 3 input 100 output network
>> x = [1; 1; 1];          % the true input
>> e = 0.1*randn(100, 1); % Gaussian noise with std = 0.1
```

```
>> y = W*x+e;              % newtwork outputs corrupted by noise
>> xhat = inv(W'*W)*W'*y   % LSQ estimate of input
      ans = 1.010
            1.0036
            0.995
```

(in practice it is more efficient and accurate to use the left-division routine `xhat = W\y` just as for the case $m = n$). Note that the inputs have been recovered to reasonable accuracy. Least squares estimation is one of the most useful techniques in applied mathematics. Expect to see much more of it.

Part 2

Reference

CHAPTER 3

Numbers and Sets

> Set theory is important ... because mathematics can be exhibited as involving nothing but set-theoretical propositions about set-theoretical entities *D M Armstrong*

1. Set Theory

A set is a collection of objects. For example we can speak of the set S containing the numbers 1, 2 and 3 which we denote by

$$S = \{1, 2, 3\}. \tag{83}$$

The order in which the elements are expressed in this notation is immaterial, so we could equally well write

$$S = \{3, 1, 2\}. \tag{84}$$

The number 3 is an element of S which we write as

$$3 \in S \tag{85}$$

and the number 4 is not S, which we write as

$$4 \notin S. \tag{86}$$

Suppose we have two sets, this times sets of letters.

$$S = \{a, b, c\} \qquad T = \{b, c, d\} \tag{87}$$

The intersection of these two sets is the set of all elements which belong to both the sets

$$S \cap T = \{b, c\}. \tag{88}$$

Their union is the set of all elements which belong to either set

$$S \cup T = \{a, b, c, d\} \tag{89}$$

(remember the set union symbol looks like the letter U for union).

It is very useful to have a notation for the empty set, that is, the set with no elements

$$\emptyset = \{\} \tag{90}$$

You should be able to work out why the following statements are true for any set S

$$S \cup \emptyset = S \qquad S \cap \emptyset = \emptyset \tag{91}$$

We often define sets by rules for membership using a notation like

$$M = \{x : x \in S \text{ or } x \in T\} \tag{92}$$

which defines the set $M = S \cup T$.

1.0.1. *Logical Symbols.* Sometimes it is useful to have a more compact notation and accurate for logical statements.

If x is a variable and P and Q are propositions Some useful constructs are

For all x

$$\forall x$$

There exists an x

$$\exists x$$

P implies Q (that is, Q is true whenever P is).

$$P \Rightarrow Q$$

P and Q

$$P \wedge Q$$

P or Q (or possibly both)

$$P \vee Q$$

Not P

$$\neg P$$

You should be able to see why the statement

$$\forall x, x \notin \emptyset$$

is necessarily true, and the statement

$$\exists x, x \in \emptyset$$

is necessarily false.

2. Number Systems

2.0.2. *The Natural Numbers.* The natural numbers are the numbers used in counting (sometimes called *cardinal* numbers).

$$\mathbb{N} = \{0, 1, 2, 3, \ldots\} \tag{93}$$

Opinion is divided as to whether 0 is a natural number. Since it is the natural answer to a "how many question" (for example how many sheep do I have, answer: zero) most mathematicians include it. If we want to leave out zero from a set we will use the star notation

$$\mathbb{N}^* = \{1, 2, 3, \ldots\} \tag{94}$$

The natural numbers are said to be *closed* under addition and multiplication since the sum and product of any two natural numbers is a natural number

$$x, y \in \mathbb{N} \Rightarrow x + y, xy \in \mathbb{N}. \tag{95}$$

They are not closed under subtraction, for example there is no number $3 - 6$ in $\mathbb{N}$. Put differently, the equation

$$x + 6 = 3 \tag{96}$$

has no solution in $\mathbb{N}$. To get a solution to this equation we need to include negative numbers.

2.0.3. *The Integers.* The integers are the whole numbers, both positive and negative

$$\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\} \tag{97}$$

The integers are closed under both addition, multiplication and subtraction

$$x, y \in \mathbb{Z} \Rightarrow x + y, xy, x - y \in \mathbb{N}. \tag{98}$$

and any equation of the form

$$x + m = n \qquad m, n \in \mathbb{Z} \tag{99}$$

now has a solution

$$x = m - n \in \mathbb{Z}. \tag{100}$$

2.0.4. *The Rational Numbers.* The integers are the fractions, both positive and negative

$$\mathbb{Q} = \{\ldots, \frac{1}{2}, -10117, 3, \ldots\} \tag{101}$$

It is clear that the brackets notation is beginning to break down. We can write instead

$$\mathbb{Q} = \{\frac{m}{n} : m \in \mathbb{Z}, n \in \mathbb{Z}^*\} \tag{102}$$

these are the fractions with any integer as numerator (top) and any integer but 0 as denominator (bottom).

Any equation of the form

$$px + q = r \qquad p, q, r \in \mathbb{Q} \qquad (p \neq 0) \tag{103}$$

now has a solution

$$x = \frac{r - q}{p} \in \mathbb{Q}. \tag{104}$$

2.0.5. *The Real Numbers.* It was a great disappointment to the ancient Greeks that the rational numbers did not seem to be adequate for geometry. For example the diagonal of a unit square has length $\sqrt{2} = 1.4142136\ldots$ which is not a rational number.

A straightforward but inelegant way of defining the real numbers is as the set of all infinite decimals, positive or negative (to make decimals unique, we need the rule that decimals ending in recurring 9 are always rounded up e.g. $0.19999999\ldots = 0.2$). We can recognise the rationals among the reals because their decimal expansions either terminate, e.g. $\frac{1}{4} = 0.25$ or recur e.g. $\frac{1}{7} = 0.142857142857\ldots$. Hence the number

$$\tau = 0.1010010001000010000001\ldots \in \mathbb{R} \tag{105}$$

(can you guess the rule for writing out τ?) is a well defined real number that is not rational.

The decimal expansion of a real number like $\sqrt{2}$ gives us the following series of ever better approximations

$$1, 1.4, 1.41, 1.414, 1.4142, 1.41421, \ldots \tag{106}$$

and this sequence of numbers lies in $\mathbb{Q}$ even if the limit, in this case, $\sqrt{2}$ does not. We can think of the infinite decimals as numbers 'plugging the holes' in $\mathbb{Q}$.

In fact this is the main reason for the importance of the real numbers, all the sequences of numbers in $\mathbb{R}$ that look as though they should have limits do have limits in $\mathbb{R}$: the set $\mathbb{R}$ is said to be *closed*.

CHAPTER 4

Notation

1. Subscripts, Superscripts, Sums and Products

Suppose we take N measurements of a length. We could call them $a, b, c, \ldots$ but this would be confusing and we would soon run out of variable names. It is convenient to use *subscript* notation and refer to the lengths as

$$l_1, l_2, l_3, \ldots, l_N.$$

The i-th measurement is then l_i. The MatLab equivalent of this is a vector `l` with elements `l(i)`.

Occasionally we use *superscripts*

$$l^1, l^2, l^3, \ldots, l^N$$

but the superscripts can be confused with exponents, for example l^2 might be read as l-squared. When confusion is likely a notation like

$$l^{(1)}, l^{(2)}, l^{(3)}, \ldots, l^{(N)}$$

(where superscripts are simply encosed in brackets) can be used.

Suppose we make M measurements on N subjects, then the i-th measurement on the j-th subject can be denoted l_{ij} using *double subscript* notation. Other notations in common use are

$$l_i^j \qquad l_{i,j} \qquad l_i^{(j)} \qquad \text{etc.}$$

MatLab implements a doubly indexed quantity like this as a two-dimensional array `l` with elements `l(i, j)`.

The sum

$$x_1 + x_2 + \ldots + x_N$$

of a N variables x_i is written using a Greek capital sigma as

$$\sum_{i=1}^{N} x_i$$

and read as 'sum from $i = 1$ to N of x_i'. This notation will often be compressed to

$$\sum_i x_i \qquad \text{or} \qquad \sum x_i \qquad \text{or even} \qquad \sum x$$

when no confusion is possible.

MatLab can evaluate the sum $y = \sum_{i=1}^{N} x_i$ of the elements of a vector `x` using a `for`-loop)

```
y = 0;              % this will store successive values of the sum
for i = 1:N         % make the list of indices to be used
   y = y + x(i);    % add in latest value of x(i)
end                 % repeat with new i until end of list
```

or directly using the single command `y = sum(x)`.

Summation signs can be used on other occasions, for example

$$\sum_{i=1}^{10} i = 1 + 2 + 3 + \ldots + 10$$

If we have a double array y_{ij} for $i = 1, \ldots, M$ and $j = 1, \ldots, N$ then we can find row sums (sum over all j keeping row i constant) and column sums (sum over all i keeping row j constant)

$$r_i = \sum_j y_{ij} \qquad c_j = \sum_i y_{ij}.$$

The sum of all the y_{ij} can be obtained as the combined total of either the row or column sums

$$\sum_i r_i = \sum_i \sum_j y_{ij} \qquad \sum_j c_j = \sum_j \sum_i y_{ij}$$

it is clear that these must produce the same result so summation signs in double sums can be commuted

$$\sum_i \sum_j y_{ij} = \sum_j \sum_i y_{ij}$$

often we simply write

$$\sum_{i,j} y_{ij} \qquad \sum\sum y_{ij} \qquad \text{or} \qquad \sum y_{ij}$$

for a double sum. In MatLab the double sum $\sum y_{ij}$ for an array `y` is obtained using `sum(sum(y))`.

Other useful facts are

$$\begin{aligned} \sum_i (x_i + y_i) &= \sum_i x_i + \sum_i y_i \\ \sum_i a x_i &= a \sum_i x_i \\ \left(\sum_i x_i\right)\left(\sum_j y_j\right) &= \sum_{i,j} x_i y_j \end{aligned}$$

you can check these formulae by writing them out in full for $i = 1, 2$, $j = 1, 2$.

There is a similar notation for the product of N variables x_i using the Greek capital pi

$$\prod_{i=1}^{N} = x_1 x_2 \ldots x_N.$$

so that

$$\prod_{i=1}^{4} i = 1 \times 2 \times 3 \times 4 = 24$$

CHAPTER 5

Functions

1. Functions in Applications

In applications functions usually turn up as the dependence of a variable (say volume of water in a bath V) on another variable (for example time since the tap was turned on t). In principle for any value of the independent variable t there is a corresponding value of the dependent variable V.

The the dependent variable is said to be a function of the independent variable and this functional dependance can be expressed explicitly as in "the volume $V(t)$" (read V of T) but also left implicit as in "let V be the volume of water at time t".

2. Functions as Formulae

An explicit formula for calculation one variable in terms of another

$$y = x^2 \tag{107}$$

is often referred to as a function. In fact this is the earliest notion of function used in mathematics. The idea is that, given a value for x, we can use the formula to evaluate the corresponding y.

3. Functions as Mappings

The general definition of function regards it as a mapping from a set S to a set T

$$f : S \to T \tag{108}$$

If we choose any element $x \in S$ then there is a unique elemnt $y \in T$ which is the image of x under f

$$f : x \to y \tag{109}$$

(read f sends x to y). which can be written equivalently as

$$y = f(x) \tag{110}$$

The set S is called the domain of f, the set T is called the co-domain. The range of f is the set

$$\text{Range}(f) = \{f(x) : x \in S\} \tag{111}$$

This range is a subset of the co-domain T but may not be the whole of T.

For example let

$$s : \mathbb{R} \to \mathbb{R}, x \to x^2 \tag{112}$$

then the range of s consists only of positive numbers

$$\text{Range}(s) = \mathbb{R}^+ = \{x \in \mathbb{R} : x \geq 0\} \tag{113}$$

Some elements of T may not be equal to $f(x)$ for any $x \in S$.

A function of variable is thus a rule for associating an output value with a given input value. The *domain* of the function is the set of allowed input values, the *range* is the set of possible output values.

We will use the notation $y = f(x)$ or $f : x \to y$ to indicate that the function f associates the input x with the output y.

A function whose domain is finite can be specified by giving a list of inputs and outputs, for example

$$\begin{aligned} F : 1 &\to 6 \\ 2 &\to 4 \\ 3 &\to 2 \end{aligned}$$

specifies a function F with domain $1, 2, 3$ and range $2, 4, 6$.

A function presented in this way can be implemented very efficiently as a look-up-table, since it involves only array access operations.

More often the functions we will deal with functions specified by a formula whose domain is the real numbers or some interval in the reals. For example

$$\begin{aligned} y &= x^3 \\ f(x) &= x^3 \\ x &\to x^3 \end{aligned}$$

all specify the same function, that which associates a number with its cube.

Mathematicians are much stricter about terminology than we will be. For example some would object to saying 'the function $f(x)$' on the grounds that $f(x)$ is merely a value of the function, not the function itself.

4. Linear and Affine Functions of One Variable

The equation of a straight line is

$$y = mx + c$$

where m is the slope (change in y for unit change in x) and c is the intercept (value of y for $x = 0$). If y is given then we have a *linear* equation for x with solution $x = (y - c)/m$. That is, we can find the inverse of the function $x \to y$ (unless $m = 0$). This is the simplest example of a linear problem. Linear problems are important because, using matrix methods, they can usually be solved just as easily as this 1D example. In this section we will begin to formalise the concept of linearity as preparation for later lectures on Linear Algebra.

Consider the case $c = 0$. The function $x \to mx$ has two crucial properties

$$\begin{aligned} m(u + v) &= mu + mv \\ m(ku) &= k(mu) \end{aligned}$$

on which we will base our definition of linearity.

We will call a function $x \to f(x)$ linear if it has the properties

$$\begin{aligned} f(u + v) &= f(u) + f(v) \\ f(ku) &= kf(u) \end{aligned}$$

In fact in one dimension the only such function is $y = mx$, since

$$y = f(x) = f(x.1) = xf(1) = mx \qquad \text{where} \qquad m = f(1)$$

where we have used the second linearity property.

A linear function always has the property that $f(0) = 0$ since

$$f(0) = f(0 + 0) = f(0) + f(0) \Rightarrow f(0) = 0$$

where we have used the first linearity property and cancelled $f(0)$'s. Thus $y = mx + c$, though it is a linear *equation* does not define a linear *function* unless $c = 0$. A function which is a linear function plus a constant term is called *affine* so $f(x)$ is *affine* only if $f(x) = g(x) + c$ where $g(x)$ is *linear*.

For functions of one variable the graph of

- an affine function is a straight line
- a linear function is a straight line through the origin

> Mathematicians often use words ambiguously if the 'correct' meaning is clear from the context. The French mathematicians Bourbaki called this useful trick *abus de notation*. For example the word linear is often used to mean affine, as in *linear equation* or *linear problem*. You just need to use a little care if you are uncertain.

5. Piecewise Specification of Functions

Sometimes a simple formula is not enough. Often different formulae must be used on different parts of the domain.

For example ocular motor neurons (OMN's) are neurons which synapse onto the muscles responsible for eye movements. As the left eye looks from left to right a neuron innervating the medial rectus muscle will increases its firing rate to contract that muscle.

Such a neuron might have the following relationship between eye position θ (measured as angle between the view direction and straight ahead in degrees) and firing rate $F(\theta)$ (measured in spikes per second)

$$F(\theta) = \begin{cases} 0 & \theta \leq 0 \\ 5\theta & 0 < \theta \leq 40 \\ 200 & \theta > 40 \end{cases}$$

This OMN does not fire for negative angles (looking to the left). Once it reaches the straight-ahead position it increases firing rate gradually from 0 to a maximum of $200\text{Hz} = 5 \times 40$ at a viewing angle of $40°$ to the right. For larger angles its firing rate remains at this maximum value.

We can implement this function directly in MatLab using the `if` statement

```
function F = firing_rate(theta)

if theta <= 0
    F = 0;
elseif theta <= 40
    F = 5*theta;
else
    F = 200;
end
```

Unfortunately this function will not work elementwise on vector inputs. To do this we need a for loop

```
function F = firing_rate(theta)
n = length(theta); % get length of vector
F = zeros(1, n);   % assign row of storage for result
for i = 1:n
    if theta(i) <= 0
        F(i) = 0;
    elseif theta(i) <= 40
        F(i) = 5*theta(i);
    else
        F(i) = 200;
    end
end
```

Plot the firing rate curve using an elementwise version of the function

```
theta = linspace(-50, 50, 100); % range -50 deg to 50 deg
F = firing_rate(theta);         % firing rate for each position
plot(theta, F);
```

(try to sketch the function before you use MatLab).

A slick vectorised version of the elementwise function is

```
function F = firing_rate(theta)

F = 5*(theta > 0 & theta <= 40)+200*(theta > 40)
```

Try to understand it (\& forms the element-wise logical *and* of two bit-vectors).

The example of the ocular motor neuron above is interesting from this point of view of invertibility. Though the input-output function is not 1-1 (for example all positions to the left are associated with output zero) it is 1-1 on the restricted domain $0 \leq \theta \leq 40$. This means that in this range we can recover eye position uniquely from the neuron firing rate (the explicit formuula is $\theta = F/5$). We say that over this range OMN firing rate *codes for position*. The position at which an OMN starts firing varies from neuron to neuron, so activity of the population as a whole specifies eye position uniquely over the whole visual range. This is called population coding.

6. Linear and Affine Functions of Two Variables

A linear function of two variables is a function $z = f(x, y)$ with the following properties

$$\begin{aligned} f(x_1 + x_2, y_1 + y_2) &= f(x_1, y_1) + f(x_2, y_2) \\ f(kx, ky) &= kf(x, y) \end{aligned}$$

In words this means

- the output for the sum of two inputs is the sum of the individual outputs
- the output after scaling the inputs equally can be obtained by scaling the original output

We can obtain a general formula for all linear functions of two variables simply by applying these rules in succession

$$\begin{aligned} f(x, y) &= f(x + 0, 0 + y) \\ &= f(x, 0) + f(0, y) \\ &= f(x.1, x.0) + f(y.0, y.1) \\ &= xf(1, 0) + yf(0, 1) \end{aligned}$$

so

$$f = ax + by \qquad \text{where} \qquad a = f(1, 0) \qquad b = f(0, 1).$$

An affine function of two variables is defined as a linear function plus a constant

$$f = ax + by + c \qquad \text{where} \qquad c = f(0, 0).$$

You should know that the graph (surface plot) of an affine function $z = ax + by + c$ is a plane and its contour lines are equally spaced straight lines.

7. The Affine Neuron

Suppose a neuron takes inputs x_1, x_2, from axons which have synaptic weights w_1 and w_2, and has a constant level of input w_0. Then the total synaptic activation is

$$a = w_1x_1 + w_2x_2 + w_0$$

which is an affine function of the inputs (compare it with the formula in the last section). Often the approximation is made that the neuronal output is proportional to this synaptic input (justified when the input is positive but not too large). This gives the affine neuron model used in many artificial neural simulations.

8. Non-Linearity and Linearisation

It is clear that linear functions are very special. Most functions are non-linear. It is important to be able to recognise linear functions because of the slogan

linear problems easy, non-linear problems hard.

Only recently have mathematicians discovered tools for attacking generic non-linear problems, and the subject of non-linear methods is still in its infancy. Even so, learning linear methods is important, since many approaches to solving non-linear problems begin by an attempt to linearise the problem.

For example some problems have fake non-linearities. If you are asked to solve the equation

$$y = 2e^x + 3$$

for y in terms of x it looks hard. However if you set $X = e^x$ the equation becomes $y = 2X + 3$, a *linear* equation for X (after which x can be obtained from X as $x = \log X$). Another example is the function

$$z = 2\sin x + 3e^y + 1$$

which is an affine function of the new variables $X = \sin x$ and $Y = e^y$.

> Linearisation by change of variable is a common technique in psychology. For example the relationship of subjective sensation to stimuli such as brightness or intensity is often linearised by a change of variable (usually a logarithm or a power law), the Weber-Fechner $\Delta I/I$ law is an example where a logarithmic change of variable can be used.

If a function is well-behaved (that is, its graph or surface plot is smooth) then for small regions that graph can be well-approximated by a line segment or a planar patch. We say the function can be decomposed into locally linear pieces. Problems can then be solved by divide and conquer strategy, in which the local linear problems are solved, and the results combined. This is the basis of many neural architectures you will meet later, such as blending function methods.

9. Using Sketches and Diagrams

Exam questions often require graphs to be plotted, curves to be sketched, etc. Often it is impossible to give good marks because simple rules are not followed. For example a satisfactory answer to 'sketch contour lines for $z = x^2 + y^2$' is not a disembodied ovalish curve floating among the text.

Using matLab means that accuracy is not a problem, but a few simple rules will help your readers. Remember to include (where appropriate)

- A title, e.g. 'Sample contour lines of $z = x^2 + y^2$'
- axes labels (simply x and y in this case)
- axis units for quantities which are not pure numbers (use the SI convention 'Height h/kg' or the more common 'Height h (kg)')
- grid lines if they make the plot clearer, or if it is likely to be used to read off values.

- marked data points to emphasise actual data used to plot the curve
- clear annotations for important features

it also helps if yo

- choose a display format for 3D data (e.g. `mesh`, `surf` or `contour`) which conveys most information rather than that which looks prettiest.
- produce all the plots at the same scale if results from multiple plots are to be compared (use the `axis` command with values from the plot with the largest data ranges).

CHAPTER 6

Calculus

1. Differentiation

1.1. Rates of Change. Let $x(t)$ be the distance (units miles) that a car has travelled along a road at time t (units hours). What do we mean by the velocity of the car at time t? Car velocities are measured here in the UK in miles per hour so the following procedure is reasonable:

Measure the position $x(t)$ at time t hours. Wait for one hour so the time is now $t+1$. Measure the new position $x(t+1)$. The total distance travelled is

$$\Delta x = x(t+1) - x(t) \tag{114}$$

and the time taken was

$$\Delta t = 1 \tag{115}$$

so the estimated velocity is the distance travelled divided by the time

$$v = \frac{\Delta x}{\Delta t} = x(t+1) - x(t). \tag{116}$$

Is this really the velocity at time t? It is clear that the velocity could have changed a lot during the measurement process, for example, the driver could have stopped and had a sandwich. At best this gives a measure of average velocity over the one hour measurement period.

What we really want is a continuous readout of velocity (similar to the one on the car's speedometer). To do this we have to take measurements spaced more closely in time. The procedure goes like this: choose a short time period Δt, measure the distance travelled

$$\Delta x = x(t+\Delta t) - x(t) \tag{117}$$

and estimate the velocity (distance per unit time) as

$$v = \frac{\Delta x}{\Delta t} = \frac{x(t+\Delta t) - x(t)}{\Delta t} \tag{118}$$

Clearly this is still an average, even if it is over a short time period. Mathematically we get round this by taking the limit as the time Δt goes to zero

$$v(t) = \lim_{t\to 0} \frac{\Delta x}{\Delta t} = \lim_{t\to 0} \frac{x(t+\Delta t) - x(t)}{\Delta t} \tag{119}$$

We will use the Newton and Leibniz notations for this quantity

$$v(t) = \dot{x}(t) = \frac{dx}{dt} \tag{120}$$

Need to add:

- derivative as slope of a curve
- example of calculating derivative as limit
- use of of derivative tables and formulae
- notation $f'(x)$

1.2. Taylor Series to First Order. If we take a short enough time Δt then the average velocity will be a good approximation to the velocity

$$\frac{dx}{dt} \approx \frac{x(t+\Delta t) - x(t)}{\Delta t} \tag{121}$$

Rearranging this formula gives

$$x(t+\Delta t) \approx x(t) + \frac{dx}{dt}\Delta t = x(t) + v(t)t\Delta t \tag{122}$$

That is, if we know the position and the velocity now we can predict (approximately) positions for a short time into the future.

Let's express in slightly different notation for the function $y = f(x)$, using the common notation

$$f'(x) = \frac{dy}{dx} \tag{123}$$

for the derivative of $f(x)$.

The approximation above becomes

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \tag{124}$$

for small h. Hence

$$f(x+h) \approx f(x) + hf'(x) \tag{125}$$

This expression is linear in the small quantity h. It is called the Taylor approximation to first (or linear) order.

- 1st order taylor as eqn of tangent line
- implications for modelling
- order of approximation
- chain rule

2. Integration

- physical interpretation of time inegral
- integral as area under curve
- fundamental theorem of integration
- definite and indefinite integrals
- using tables of integrals and formulae

CHAPTER 7

Special Functions

> ... special functions provide an economical and shared culture analogous to books: places to keep our knowledge in, so that we can use our heads for better things. *Michael Berry*

1. The Straight Line

We begin with the well-known function

$$y = mx + c. \tag{126}$$

As the following code segment shows this the equation of a straight line

```
x = linspace(-1, 1, 100);
m = 1/2; c = -1;
y = m*x+x;
plot(x, y)
```

with slope m and intercept c.

This function is invertible if $m \neq 0$ giving

$$x = \frac{y-c}{m} = \frac{1}{m}y - \frac{c}{m}. \tag{127}$$

which is again the equation of a straight line.

> By analogy an expression is called linear in a variable x if it can be written as
> (terms not depending on x) + (terms not depending on $\times x$

2. The Quadratic

The simplest non-linear function is obtained by including quadratic terms

$$y = ax^2 + bx + c. \tag{128}$$

In the degenerate case $a = 0$ we are back to the straight line. Otherwise the graph of this equation is a parabola. If $a > 0$ (bowl full) its unique stationary point which is a minimum, if $a > 0$ (bowl empty) it is a maximum.

The function does not have a unique inverse but there is a formula giving the two possible values, since we can solve the quadratic equation

$$ax^2 + bx + (c - y) = 0 \tag{129}$$

using the standard formula to get

$$x = \frac{-b \pm \sqrt{b^2 - 4a(c-y)}}{2a} \tag{130}$$

The derivative is the linear function

$$\frac{dy}{dx} = 2ax + b \tag{131}$$

and second derivative is a constant

$$\frac{d^2y}{dx^2} = 2a, \tag{132}$$

all higher derivatives are zero.

The fact that the derivative of a quadratic is linear means that we can easily solve for the position of the stationary points. From $\frac{dy}{dx} = 0$ we find a single stationary point

$$x_0 = \frac{-b}{2a} \tag{133}$$

If $a > 0$ this is a global maximum, if $a < 0$ it is a global minimum, so it is always an extremum.

The value of the function at the extremum is

$$x_0 = c - \frac{b^2}{4a}. \tag{134}$$

A non-calculus way to obtain this result is by completing the square

$$y = a(x - \frac{b}{2a})^2 + (c - \frac{b^2}{4a}). \tag{135}$$

and staring at the result.

It is the ease with which maxima and minima can be located which makes quadratic expressions so important in applied mathematics (see: the Lamp Post Factor).

3. The Exponential Function

The exponential function is denoted by

$$y = e^x = \exp(x) \tag{136}$$

It's inverse is the logarithmic function (with base e)

$$x = \ln(y) \qquad e^{\ln(y)} = y \tag{137}$$

It has the everywhere convergent Taylor expansion

$$y = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{n=1}^{\infty} \frac{x^n}{n!} \tag{138}$$

which can be used to define this function.

It's most important property is that it is it's own derivative

$$\frac{de^x}{dx} = e^x \qquad \frac{dy}{dx} = y. \tag{139}$$

and hence it's own integral

$$\int e^x dx = e^x + c \tag{140}$$

It converts summation into multiplication

$$\exp(x + y) = \exp(x) \exp(y) \tag{141}$$

Powers of another number p can be written in terms of the exponential function

$$y = p^x = (e^{\ln(p)})^x = e^{x \ln(p)} \tag{142}$$

It is useful to memorise the slightly more general form of the derivative

$$y = e^{ax} \tag{143}$$

$$\frac{dy}{dx} = ay. \tag{144}$$

4. The Logarithmic Function

$$y = \ln(x) \tag{145}$$

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} + \dots \tag{146}$$

$$\frac{dy}{dx} = \frac{1}{x} \tag{147}$$

$$y = \log_a(x) = \frac{\ln(x)}{\ln(a)} \tag{148}$$

$$\ln(xy) = \ln(x) + \ln(y) \tag{149}$$

5. Basic Trigonometric Functions

$$u = \sin(t) \tag{150}$$

$$\frac{du}{dt} = \cos(t) \tag{151}$$

$$\int sin(t)dt = -\cos(t) + c \tag{152}$$

$$\sin(t + 2\pi) = \sin(t) \qquad \pi = 3.1415926535\dots \tag{153}$$

$$\sin(t + \pi) = \cos(t) \tag{154}$$

$$sin(\omega t) \tag{155}$$

$$sin(2\pi f t) \tag{156}$$

$$T = \frac{2\pi}{\omega} = \frac{1}{f} \tag{157}$$

6. Basic Hyperbolic Functions

Part 3

MatLab

CHAPTER 8

MatLab: a Quick Tutorial

We will be programming in MatLab, a very popular environment for mathematical modelling. Because it was designed for research and prototyping of scientific and engineering applications it is fast and flexible. It has many built-in features which greatly simplify the writing of modelling software

- a fairly standard interface across different platforms (include PC's, Mac's and Unix machines)
- good graphics/visualisation capabilities
- many high-level mathematical operations implemented as standard
- extensive toolboxes (such as the image processing toolbox) which extend its capabilities

MatLab is used extensively within the Department of Psychology where current applications of MatLab include

- modelling the control functions of the basal ganglia
- modelling cerebellar control of eye movements
- producing movies for visual psychophysics
- analysing psychophysical data on stereo vision
- analysing sequences of brain imaging data (optical and fMRI)
- investigating neural architectures for signal analysis.

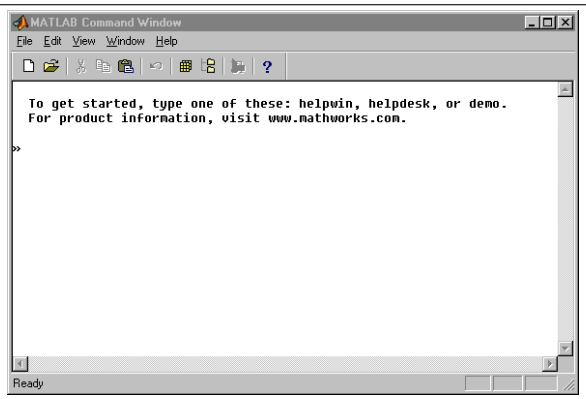
1. Entering Commands and Correcting Errors

Running (opening/launching) MatLab produces a command window as shown in Figure 1. Matlab is an interactive language - you type in expressions and MatLab evaluates them and tells you the answer. The prompt sign:

>>

shows that MatLab is waiting for input from you.

Figure 1 The MatLab Command Window as seen on a PC.



You may have to click on the command window before you can start typing. After you have typed in an expression to evaluate MatLab waits for you to hit the return key before it starts processing your commands.

You can correct typing errors on the command line on a PC or Mac in the usual way, using the editing keys:

- left and right arrow keys to move the cursor (insert point) left or right
- delete/backspace to forward/backward delete characters
- the mouse to cut and paste text from elsewhere
- the up-arrow key to retrieve previous commands (you can use up and down-arrows to navigate through the entire history of previous commands)

alternatively confirmed hackers can use emacs-style keyboard editor commands if they prefer.

If a command is incorrect MatLab will usually display an error message. It is worth remembering that some mistakes may lead to long calculations and the machine will stop responding, if this happens the calculation can (sometimes) be interrupted using Ctrl-C (press Ctrl and C together).

2. MatLab as a Desk Calculator

We will begin by using MatLab to do some simple arithmetic. After the MatLab prompt type

```
1+1
```

then hit return. MatLab will reply with something like

```
ans = 2
```

Here are some other examples (you need not enter the explanatory comments which appear after the \% sign):

```
>> 3+2       % this comment is ignored
   ans = 5
>> 3-2
   ans = 1
>> 3*2       % the * denotes multiplication
   ans = 6
>> 7/2       % the / denotes division
   ans = 3.5
>> 3^2       % 3 squared (a^b means a to the power of b)
   ans = 9
```

> The result $7/2 = 3.5$ may surprise those with experience of other computer languages where *integer* division would be used so that $7/2 = 3$ (the remainder 1 being ignored). By contrast all numbers in MatLab are stored as double precision floating point so 7/2 means 7.0/2.0 and evaluates to 3.5. A little thought (or knowing how many almost untraceable programming errors the other convention has generated) will convince you that MatLab got it right.

Parentheses can be used in MatLab expressions to group expressions and make the order of evaluation clear

```
>> 3*(2+2)       % addition first
   ans = 12
>> (3*2)+2       % multiplication first
   ans = 8
```

When parentheses are not present expressions are evaluated in a strict order of precedence. Details can be found in the MatLab manual. For example in

```
>> 3*2+2
   ans = 8
```

the multiplication is performed before the addition. If the order of evaluation is not completely obvious it is good practice to insert parentheses to make clear the order you want.

3. Variables

MatLab allows the use of variables (a bit like memories on a calculator) to which values can be assigned and which can be used in expressions.

```
>> a = 3    % store the value 3 in a variable called 'a'
   ans = 3
>> b = 4    % store 4 in b
   ans = 4
>> a+b      % add the value stored in 'a' to that in 'b'
   ans = 7
```

Variable names can be short, such as `x`, or meaningful, for example `muscle_tension`, according to taste. They cannot contain spaces or any of the special characters (such as $*$) used by MatLab. It is best not to use names already in use by MatLab (such as `ans`).

Some variables have values pre-assigned. For example `pi`, an important number you may have heard of, is preset to $\pi = 3.14159....$

You can get information about all the variables you have currently defined using `whos`. To clear all the current user definitions type `clear all`, this also frees up the associated memory.

4. Punctuating MatLab

Ending a line with a semi-colon (;) suppresses any MatLab output for that command. This can be used to stop MatLab printing unwanted values of intermediate expressions in a calculation

```
>> a = 10; % don't print ans = 10
>> b = 5;  % don't print ans = 5
>> a/b     % do want this result
   ans = 2
```

It is probably a good idea to put a semi-colon after any expression whose output is irrelevant or which might take too long to display on screen, for example typing something as innocent as

```
zeros(100)
```

prints 10000 zeros to the screen.

Semicolons can be used to separate expressions on the same line

```
>> a = 3; b = 4; a^b
    ans = 81
```

defines a and b and calculates a^b, printing only the final result. Alternatively commas can be used to separate expressions when you need to see intermediate results:

```
>> a = 3, b = 4, a^b
    a = 3
    b = 4
    ans = 81
```

Using a semi-colon by mistake will mean that no output is printed. For example after typing

```
>> 3*4+5;
```

MatLab performs the calculation but does not tell you the answer. This can sometimes be frustrating after a long calculation, however MatLab always stores the result of the *last* calculation in a variable called `ans` so that it can be accessed and used again

```
>> 3*4+5; % whoops! forgot to print result
>> ans    % saved!
    ans = 17
```

5. Row Vectors in MatLab

MatLab is designed to make calculations involving vectors and matrices easy For the moment by a row vector we simply mean a list of numbers written in a row; for example $v = (5.0, 4.1, 3.2, 2.3, 1.4)$ is a row vector with 5 components.

To make a row vector in MatLab type in a list of numbers enclosed in square brackets. For example

```
>> vec = [1 2 3 4 5 6]
    vec = 1 2 3 4 5 6
```

creates a 6-component row vector (or 6-vector) called `vec`. Note that terminology varies, for example when a vector is thought of as a list it is common to refer to its entries or elements rather than to its components.

Values of individual components of a vector can be recovered by using bracket notation: for example the second component of `a` is `a(2)`, the components of a vector can be used in calculations as follows

```
>> a = [4 5 6];  % make 3-component row vector
>> a(2)          % second component of  a
    ans = 5
>> a(1)+a(3)     % sum of first and third components
    ans = 10
>> a(4)          % a has no fourth component - gives Error message
    ???  Index exceeds matrix dimensions
```

> Indexing (numbering of array elements) in MatLab starts at 1 because it is based on FORTRAN rather than C or C++ (whose array indices start at 0).
> Note that MatLab Vectors do not need to be defined, declared or allocated as in C. They are allocated on request and memory is reclaimed when the variable is re-assigned. Anyone who has programmed maths/modelling code in C (or worse still in Pascal or Modula2) will appreciate how much programming tedium is avoided by an interactive, (almost) typeless language with dynamic allocation.
> This prompts a short explanation for the programming purity police who sniff at the lack of 'safety features' in MatLab: Mathematical modelling in the research community usually involves relatively small programs which are subject to frequent revision - usually by their original authors. Often these programs will only be run once in their final form. Because of this MatLab can afford to ignore the tedious restrictions which are imposed on programmers by other popular computer languages in order to make it possible for mediocre programmers to maintain over-elaborate code long past its delete-by date.

MatLab allows arithmetic operations on vectors. For example

```
>> [2 3 4 5]+1          % add 1 to each element
   ans = 3 4 5 6
>> [2 3 4 5]*2          % double each element
   ans = 4 6 8 10
>> [2 3 4 5]/2          % halve each element
   ans = 1 1.5 2 2.5
```

If vectors are the same length they can be added and subtracted elementwise

```
>> [2 4 6]+[2 3 4]
    ans = 4 7 10
>> [2 4 6]-[2 3 4]
    ans = 0 1 2
```

Multiplication, division and powers have to be specified slightly differently for vectors than for scalars (ordinary numbers), an extra full stop is needed before the operation symbol

```
>> [2 4 6].*[2 3 4] % elementwise product
    ans = 4 12 24
>> [2 4 6]./[1 2 3] % elementwise quotient
    ans = 2 2 2
>> [1 2 3].^2       % elementwise square
    ans = 1 4 9
>> [1 2 3].^[1 2 3] % elementwise power
    ans = 1 4 27
```

Important note: The extra dot specifies that the operation be performed elementwise, that is, applied to each pair of corresponding elements in order.

> MatLab reserves the simpler notation without a dot for the matrix/vector versions of these operations which will be discussed later.

5.1. Making Row Vectors. Row vectors filled with zeros or ones can be made using the commands `ones` and `zeros`

```
>> a1 = zeros(1, 6)       % one row of 6 zeros
    ans = 0 0 0 0 0 0
>> a2 = ones(1, 1000000); % good idea to suppress this output
```

The are two very useful commands to make vectors with equally spaced elements. The two-argument colon operator `a:b` produces unit steps going from `a` to `b`

```
>> 1:4
    ans = 1 2 3 4
>> 3:8
    ans = 3 4 5 6 7 8
```

The three-argument colon `a:b:c` makes a vector starting at `a` going in steps of `b` to a maximum of `c`

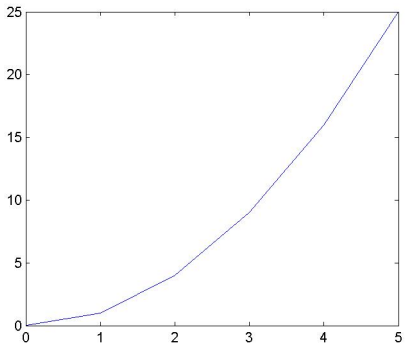
```
>> 1:2:10
    ans = 1 3 5 7 9    % note 10 is missing
>> 4:-1:1
    ans = 4 3 2 1      % use negative steps to go down
```

The function `linspace(a, b, n)` makes a vector starting at `a`, ending at `b` with `n` equally spaced elements

```
>> linspace(0, 1, 5)    % 5 elements from 0 to 1
   ans = 0 0.25 0.5 0.75 1
>> linspace(1, 5, 5)    % same value as 1:5
   ans = 1 2 3 4 5
```

6. Plotting Graphs Using MatLab

Figure 2 MatLab plot of the function $y = x^2$.



Suppose we want to plot the graph of $y = x^2$ over the range $0 \leq x \leq 5$. To do this by hand we would select a few values for x in the given range, calculate the corresponding y's by squaring, then plot these points and draw a smooth curve through them. The procedure in MatLab is very similar.

First make a vector of equally spaced x values using the colon operator

```
>> x = 0:5
   ans = 0 1 2 3 4 5
```

then square all these values to get the corresponding y-values

```
>> y = x.^2           % note the elementwise .^
   ans = 0 1 4 9 16 25
```

To plot y as a function of x we use the `plot` command

```
>> plot(x, y)
```

A figure window will appears containing the graph shown in Figure 2.

Note that MatLab does not draw a smooth curve through the plotted points, it simply joins them up using straight lines. To get a smoother curve we need to plot many points, though tedious by hand, this is straightforward using MatLab:

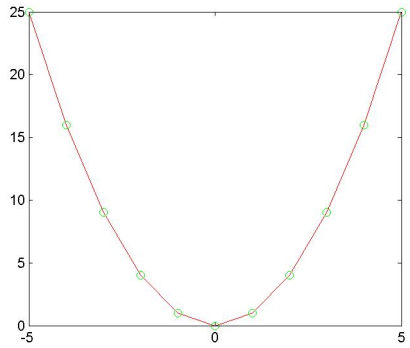
```
>> x = linspace(0, 5, 100); % 100 equally space points
>> y = x.^2;
>> plot(x, y);
```

(note the use of semicolons to suppress output of long intermediate results).

There are many options to alter the way data is plotted. Figure 3 was produced by the following code

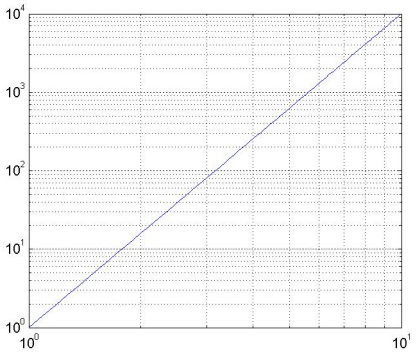
```
>> x = -5:5; y = x.^2;
>> % plot red graph and overlay green circles
>> plot(x, y, 'r', x, y, 'go')
```

Figure 3 A more elaborate MatLab plot of $y = x^2$.



(for more information about plot options type `help plot`).

Figure 4 A log-log plot of $y = x^4$ gives a straight line.



Sometimes it is useful to plot graphs using logarithmic axes (for example when a positive variable varies over a very large range, or to recognise special relationships such as power laws). MatLab provides special plot functions for this purpose:

```
>> x = linspace(1, 10, 100); y = x.^4;
>> figure(1); plot(x, y)          % usual plot
>> figure(2); semilogx(x, y)      % log scale on x-axis
>> figure(3); semilogy(x, y)      % log scale on y-axis
>> figure(4); loglog(x, y)        % log scale on both axes
```

the last line produces the log-log plot shown in Figure 4.

A shortened form of the plot command takes just one argument y, in this case the values on the x-axis are the data indices $(1, 2, 3, 4, \ldots)$ so that the following plot commands are equivalent:

```
>> plot([7 6 5 6 7]);
>> plot([1 2 3 4 5], [7 6 5 6 7]);
```

Pieces of text are called strings and are enclosed in single quotes to stop MatLab trying to evaluate them. Text information can be added to a graph in various ways, for example

```
>> title('Ocular␣Muscle␣Length-Tension␣Curve'); % Add text as title
>> xlabel('length/mm');                         % label for x-axis
>> ylabel('tension/N');                         % label for y-axis
```

```
>> grid                                    % add grid lines
```

If a graph is worth saving or printing always give it a title and label the axes. This is a matter of personal hygiene for scientists.

> Matlab text strings are enclosed in single quotes unlike C. They are are actually vectors of ASCII code numbers for the individual characters. MatLab flags them to indicate that they must be output as text strings, but they can be also be used in arithmetic expressions, just like other vectors:
>
> ```
> >> 'abcd'+0
>     97    98    99   100    % gives ascii codes for a b c d
> ```

Multiple plots on the same axes can be produced in many ways. One way is to prevent MatLab from clearing the old plot before drawing the new using `hold`

```
>> x = linspace(0, 1, 100);
>> plot(x, x.^2, 'r');      % plot x-squared in red
>> hold on                  % hold figure
>> plot(x, x.^3, 'g');      % overlay plot of  x-cubed in green
>> hold off                 % release figure
```

Alternatively `plot` can be given multiple plot lists. In this case an explanatory legend can be attached

```
>> plot(x, x.^2, 'r', x, x.^3, 'g'); % same effect as above
>> legend('square', 'cube');         % labels for graphs
```

The axis limits for a plot are chosen automatically so that all the data points given will fit. The limits can be set manually using the function `axis([xmin xmax ymin ymax])`. This command can be used to make a plot more presentable, force different plots to have the same axes, or to zoom in on part of a plot.

7. Script M-Files

Typing multi-line commands into MatLab without making mistakes can be tedious. Scripts allow multiple commands to be edited and saved in a file which will then be run exactly as if they had been typed in directly. Scripts always have names with a `.m` extension, such as `foo.m`.

Choose *New M-File* from the *File* menu. A simple text-editor window will appear into which you can type the following commands

```
x = linspace(0, 12, 100);
y = sin(x);
plot(x, y);
```

Using *Save-as* on the *File* menu put the file either into public space, or into your own folder, with a name ending in `.m` such as `example.m` (do not use spaces in the filename). In your MatLab command window you can now type the filename (without the `.m`)

```
>> example
```

and the plot will appear.

Scripts behave as though the commands had been typed into MatLab directly, so variables used in the script (such as x and y in the example) are still defined when it terminates, and can be used in subsequent commands, and values of variables previously defined outside the script can be used without passing them as arguments. Because of this scripts are often regarded as dangerous, and later we will learn to use function M-files to avoid these problems. However the extra access to variables given by scripts is often very useful, especially during code development.
MatLab is well adapted to a variety of programming styles including structured-imperative, functional, and even object oriented styles. This leaves the choice of programming style where it should be, in the hands of the programmer.

8. The MatLab `for`-Loop

A loop structure allows commands to be evaluated repeatedly. The general form of the MatLab `for` loop is

```
for i = list-of-values
   do-this
   do-that
   do-t'other
   ...
end
```

This tells MatLab to repeat all the commands between the `for`-line and the `end`-statement using successive values of `i` from the `list-of-values`. The indentation of the lines between `for` and `end` is not mandatory, but helps with clarity and is regarded as good style.

Many of you will have met similar loop-control structures in other languages, however if any of the examples below are unclear *ask for help.*

To get MatLab to print hello once we can issue the display command

```
>> disp('hello')
    hello
```

To print hello 100 times type in these three lines of code

```
>> for i = 1:100
      disp('hello')
   end        % MatLab does nothing till it sees this end statement
```

Multi-line structures like this are often entered on a single line as

```
>> for i = 1:100, disp('hello'), end
```

with the commands separated by commas.

The following examples require more typing and should be run using a script M-file.

Find the sum of the numbers from 21 to 506:

```
tot = 0;            % this will store successive values of the sum
for i = 21:506      % make the list of numbers to be added
   tot = tot + i;   % add in latest value of i
end                 % repeat with new i until end of list
disp(tot)           % display the total
```

(why was the name `tot` used rather than the more obvious `sum`?).

Print all multiples of 3 less than 50

```
for i = 3:3:50      % numbers starting at 3 in steps of 3 up to 50
   disp(i);         % print latest value of i
end                 % repeat with new i until end of list
```

9. MatLab Functions

We have already met some MatLab functions, For example `linspace` and `plot`. Another example is `sum`, which adds up all the elements in a vector

```
>> sum([1 2 3 4])
    ans = 10
```

MatLab has many built-in mathematical functions such as the trigonometric functions `sin`, `cos` and `tan` and logarithms and exponentials, `log` and `exp`. It has functions for statistics such as `mean` and `std` to calculate mean and standard deviation. The MatLab toolboxes contain thousands of functions for more specialised applications.

Interpreted languages like MatLab are usually slower than compiled languages such as C. However many MatLab applications run nearly as fast as optimal C-code. This is because most built-in functions and operations such as `+`, `-`, `sum`, etc. are in fact compiled functions written in C. MatLab calls code using these fast constructs 'vectorised' (misleadingly since it suggests parallel implementation). The availability of fast code in an interpreted environment is one of the main advantages MatLab has over competitors such as Mathematica.

Chunks of code can be timed using `tic` and `toc`. Try comparing an interpreted loop structure with equivalent vectorised code by running the script below substituting various values of `n`

```
n = 10000;
tic              % start timer
tot = 0;
for i = 1:n;
    tot = tot+i; % sum numbers 1 to n using for-loop
end
toc              % end timer and print loop time

tic              % re-start timer
tot = sum(1:n);        % sum numbers 1 to n using built in function
toc              % end timer and print vectorised time
```

(why does the vectorised code show increased benefit for larger `n`?).

Exercises

When possible evaluate expressions by hand before checking the answers in MatLab. Don't worry if you have not met some of the ideas - just ask for help.

(1) Evaluate

```
1+(2*3)          4/(2*2)
(4/2)*2          3^(1+2)
```

(2) Evaluate

```
1+2*3            4/2*2            8/4/2
1+2/4            2*2^3            2^2/3
```

Try to determine in what order MatLab peformed the operations.

(3) Explain and evaluate the code segment below (`sqrt(x)` is the square root $\sqrt{x}$)

```
o = 3 ; a = 4;
H = sqrt(o^2+a^2)
A = o*a/2
```

(4) Create the vectors `x = [3 1 4 1 5 9 2 6 5]`, `y = [1 2 3]` and evaluate

```
x(6)/y(3)
x(1)^y(2)
x(10)
[x(1) x(2) x(3)].*y
```

(5) Make 10 linearly space values between 50 and 51 using `linspace`. Why is this easier than using the colon operator?
(6) Plot graphs of the following functions
   (a) $y = x^3$ for $0 \leq x \leq 1$
   (b) $a = \sin t$ for $0 \leq t \leq 10\pi$ (use the MatLab variable `pi` and built in function `sin`)
   (c) $y = e^x$ for $-3 \leq t \leq 3$ (to get e^x use the built in function `exp(x)`)
(7) Plot $y = x^2$ for $0 \leq x \leq 1$ and overlay a plot of $y = 1 - x$ using `hold`. How can the plot be used to find an approximate solution of $x^2 = 1 - x$? Find such a solution and check that it works.
(8) Guess what `rand` does by plotting its output as shown below

```
r = rand(100, 1);
plot(r, 'go');
```

(9) For variables $x_1, \ldots, x_M$, $y_1, \ldots, y_N$ use summation notation to write down (including summation limits)
   (a) the sum of squares of the x's
   (b) the sum of products of the x_i's and their indices i.
   (c) the sum of all products of the x_i's and the squares of the y_j's.
(10) Write a script to sum the cubes of numbers from 1 to 1000 using a `for` loop. Check the result using `sum`.
(11) Calculate $\sum_{i=1}^{3} \sum_{j=1}^{2} ij^3$.
(12) Suppose $\sum x_i = 0$. What is $\sum x_i y_j$?
(13) Why is $2^{\sum x_i} = \prod 2^{x_i}$?
(14) Run the MatLab `demo` to see some other stuff.

10. More Maths and MatLab

In this chapter we will learn some more MatLab syntax. and then look at some terminology and definitions associated with functions of one and two variables.

11. Function M-Files

The MatLab built-in function `sum` evaluates and returns the sum of a list of numbers:

```
>> x = [1 2 3 4]; s = sum(x)
    s = 10
```

The vector `x` is called the *input* or *argument* of the function `sum`, which returns the sum of the elements of the vector as its *output* or *value*.

We can write our own functions to perform calculations and return values. Open a new M-File containing the code

```
function s = sumsq(x)
s = 0;            % initialise s to zero
n = length(x);    % get length of list
for i = 1:n       % i steps from 1 up to n
    s = s+x(i)^2  % the square of each x(i) is added into s
end
```

and save it as `sumsq.m`.

This function calculates the sum of the squares of the elements of the input vector. The first line of the M-file specifies that the file `sumsq.m` contains a function definition (as opposed to a script), that the function take one input argument (to be called `x`) and that it returns one output value (to be called `s`. The remaining lines specify how `s` is to be calculated from `x`.

A function you have defined in this way can then be called from the MatLab session just like a built in function

```
>> a = [1 2 3 4]; tot = sumsq(a)
    tot = 30
```

Note that when *calling* the function we can use any names we wish for the argument and returned value.

> Variables used inside functions are not available for use outside the function, and those defined outside are not available inside the script (even if they have the same name). This means that programming using functions is safer than programming with scripts, since the correctness of a function is not dependent on the correctness of the program in which it is embedded. Since in addition MatLab functions cannot modify their arguments (without tricky programming using the foreign function interface), MatLab is to some extent a *functional* language, which earns it Brownie points with purists.

MatLab functions can have any number of inputs. For example we might define a new plot function with two arguments

```
function fancyplot(x, y)
figure(1); plot(x, y, 'r', x, y, 'go');
```

which plots data into figure window 1 as a continuous red curve with the data points highlighted by green circles. This function has two inputs and no outputs.

Functions can also have multiple outputs. For example some simple descriptive statistics can be recovered by the function

```
function [m, s] = stats(x)
n = length(x);              % number of elements of vector x
m = sum(x)/n;               % mean
```

```
s = sqrt(sum((x-m).^2)/n);   % standard deviation
```

which has one input, a data vector, and returns two values, the mean and standard deviation of the data. Note the very simple syntax for specifying that two values are returned by the function.

When a function returns multiple values not all the outputs need to be assigned. For example the function `stats` can be called in three ways

```
>> x = [1 -1];          % very simple input
>> [m, s] = stats(x)    % (1) return two values and assign to m and s
    m = 0
    s = 1
>> m = stats(x)         % (2) return one value and assign to m
    m = 0
>> stats(x)             % (3) return no value (but set ans)
    ans = 0
```

Documenting code is tedious but essential. MatLab has a nice way of including help information in functions which makes it instantly accessible to users. If we include a continuous block of comments immediately after the function declaration then typing `help function-name` in the MatLab session will print out the comment block, for example the helpful comments in

```
function [m, s] = stats(x)

% function [m, s] = stats(x)
% Calculates mean and standard deviation
% x = input data vector
% m = data mean
% s = data standard deviation

n = length(x); m = sum(x)/n; s = sqrt(sum((x-m).^2)/n);
```

can be recovered by typing

```
>> help stats

    % function [m, s] = stats(x)
    % Calculates mean and standard deviation
    % x = input data vector
    % m = data mean
    % s = data standard deviation
```

I like to include in the help comment a copy of the function declaration (which shows how to call the function), a short description of the computation it performs, and a description of the format of the input and output arguments.

12. Elementwise Functions

If MatLab code for a function of with scalar inputs is written carefully it can be used to calculate outputs corresponding to a vector of inputs. For example the implementation

```
function y = fun(x)
y = 1+x^2;
```

of $y = 1 + x^2$ works OK for single inputs

```
>> fun(3)
    ans = 10
```

but fails if we supply a list of inputs

```
>> fun([0 1 2 3])
    Error !***???!
```

Howver a small change to the code makes it work,

```
function y = fun(x)
y = 1+x.^2;
```

we just needed an *element-wise* square inside the function. Now our function works element-wise

```
>> fun([0 1 2 3])
    ans = [1 2 5 10]
```

just like built-in functions such as `sin`.

13. The MatLab `if` Statement

The `if` statement allows certain actions to be performed only if associated conditions are satisfied. It has the general form

```
if     condition_1
    action__1
elseif condition__2
    action__2
...
elseif condition__N
    action__N
else
    another_action
end
```

(the `else` option need not be present). For example

```
if pi < 22/7
    disp('pi␣<␣22/7')
else
    disp('pi␣>=␣22/7')
end
```

will check whether π is bigger or smaller than the common approximation $22/7$ and display the result.

We can simulate the tossing of a coin using the MatLab random number generator `rand` which returns a random number in the range 0 to 1 and the `if` statement. Half the time this number will be less than 0.5 and half greater. We can make a list of 100 simulated coin tosses with the following code

```
tosses = [];    % empty list
for i = 1:100
    r = rand(1);
    if r < 0.5
        tosses(i) = 0; % heads
    else
        tosses(i) = 1; % tails
    end
end
```

(why does `sum(tosses)` tell us how many tosses came out tails). Random (strictly speaking pseudo-random) data generated in this way is basic to the computational technique known as Monte-Carlo simulation.

There is a fast vectorised version of the code above. We can make 100 random numbers directly using `r = rand(1, 100)`. Simulated coin tosses can then be constructed using `tosses = (r $>$ 0.5)`. This produces a bit-vector (vector of noughts and ones) with zeros where $r > 0.5$ is false and ones where it is true. Many potentially slow MatLab `for-if` combinations can be avoided using logical operations and bit-vectors in this way.

14. One-to-one Functions and their Inverses

A function is said to be one-to-one (1-1) if any given output is associated with just one input value. This property can depend on the domain of the function. For example $f(x) = x^2$ is 1-1 on the domain $0 \leq x \leq \infty$ because no two positive numbers have the same square. However on the domain $-\infty \leq x \leq \infty$ it is many-to-one since, for example 2 and -2 both have the same square $4 = (-2)^2 = 2^2$.

When a function $f : x \to y$ is 1-1 it has an inverse function called f^{-1}. This associates the outputs with the unique inputs to which they correspond f^{-1} : $y- > x$. The inverse of $f(x) = x^2$ on the domain $0 \leq x \leq \infty$ where it is 1-1 is $f^{-1}(x) = \sqrt{x}$.

Finding inverses of functions is important. One approach to modeling cognition is to present it as an inverse problem. Our sense organs take the surrounding world as input, transform it into nerve impulses, and send it to the brain as output. The the task of the brain is to invert this process and recover the useful aspects of the structure of the world from the nerve impulses. Much computational research uses this paradigm, for example the traditional AI approach to vision was thoroughly re-constructionist. There are other important approaches, for example research which trains a generic neural net to perform a range of tasks given some form of input data is behaviourist in style.

15. Matrices in MatLab

A matrix is a rectangular array of numbers. In MatLab matrices can be constructed by specifying the elements of the matrix, enclosed in square brackets, the rows being separated by semi-colons

```
>> mat = [1 2 3; 4 5 6; 7 8 9]
   ans = 1 2 3
         4 5 6
         7 8 9
```

A matrix with m rows and n columns is called an $m \times n$ matrix. The element in the `i`-the row and `j`-the column of a matrix can be accessed as `m(i, j)`

```
>> mat(2, 3)
    ans = 6
```

In mathematical work we will usually use the subscript notation M_{ij} for the (i, j)-th element of a matrix M.

All the arithmetic operations can be applied to matrices as well as vectors, for example

```
>> mat.^2          % square all the elements of mat
   ans = 1   4  9
        16 25 36
        49 64 81
```

A column vector is a matrix with only one column (an $m \times 1$ matrix)

```
>> v = [1; 2; 3]
    ans = 1
          2
```

```
           3
```

The row vectors we have already met are $1 \times n$ matrices.

Matrices whose elements are all the same can be constructed using `zeros` and `ones`

```
>> z = zeros(2, 3)        % 2 by 3 of zeros
   z = 0 0 0
       0 0 0
>> a = ones(1, 4)         % 1 by 4 of ones
   z = 1 1 1 1
>> a = 4.5*ones(2, 2)     % 2 by 2 of 4.5's
    a = 4.5 4.5
        4.5 4.5
```

Suppose we want to perform some operation on every element of a matrix. The operation will have to be performed on the element in each column j of a given row for every row i. Such a sequence of operations can be specified by a *nested* for-loop. The following code prints every element in a matrix `A`.

```
[m, n] = size(A);      % returns two values - no. of rows m, and no. of cols n
for i = 1:m            % for each row
    for j = 1:n        % for each col
        disp(A(i, j))  % display that element
    end
end
```

Note the use of indentation to clarify the structure of the nested loop.

16. Functions of Two Variables

A function of two variables is a rule that associates an output value with *two* input values

$$f : (x, y) \to z$$

or $z = f(x, y)$. Such functions will often be presented as formulae such as

$$z = x^2 + y^2$$

which show how to calculate the output z from x and y.

A matrix can be thought of as a two dimensional look-up table for the function associating the input pair (i, j) with the element in the i-th row and j-th column of a matrix

$$(i, j) \to M_{ij}$$

17. Visualising Functions of Two Variables

If we use x and y as coordinates in the plane, and interpret z as height above that plane, a formula like $z = x^2 + y^2$ defines a surface. Plotting this surface is a useful way to present a function of two variables visually.

We want to specify the heights z at a number of points x, y. It is simplest if these points form a grid. We make such a grid by specifying the spacing we want in x and y then using the function `meshgrid` to make the grid. This is easiest to explain using an example

```
>> x0 = linspace(-2, 2, 5) % 5 values from -2 to 2
  x0 = -2 -1 0 1 2
>> y0 = linspace(-1, 1, 3) % 3 values from -1 to 1
  y0 = -1 0 1
```

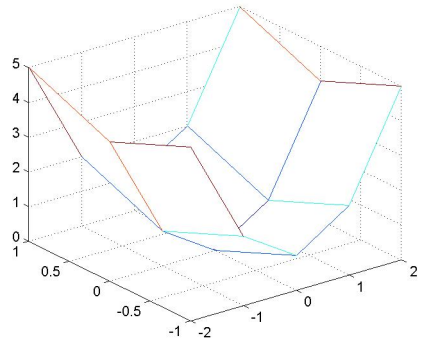
```
>> [x, y] = meshgrid(x0, y0) % make mesh with given spacing

   x = -2 -1  0  1  2
       -2 -1  0  1  2
       -2 -1  0  1  2

   y = -1 -1 -1 -1 -1
        0  0  0  0  0
        1  1  1  1  1
```

The result is two matrices of the same dimensions, one varying only along its rows, one down its columns. Each matrix specifies either the x or y coordinates of points in a 3×5 grid. The associated height z can be calculated by translating the formula $z = x^2 + y^2$ into MatLab

Figure 5 A simple mesh plot of 3D data.




```
>> z = x.^2+y.^2  % dot means elementwise
    z = 5 2 1 2 5
        4 1 0 1 4
        5 2 1 2 5
```

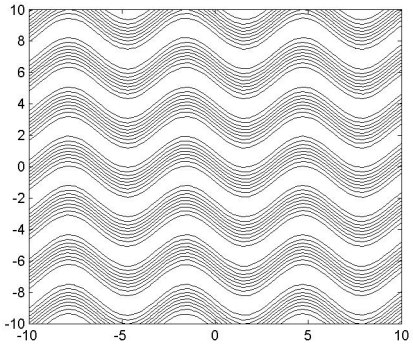
The surface can then be plotted using

```
>> mesh(x, y, z); % draw surface as a wire mesh
```

to give Figure 5.

Figure 6 shows the more complex function

Figure 6 A fine mesh plot of the function $z = \sin(y + \sin x)$



$$z = \sin(\sin(x) + y)$$

plotted on a finer grid using the following script

```
l = linspace(-10, 10, 50);
[x, y] = meshgrid(l, l);   % square grid with same x and y spacing
z = sin(sin(x)+y);
mesh(x, y, z);
```

> The surface is drawn from a default view position. This can be changed using the command `view(az, el)` where `az` and `el` specify the view position in terms of its azimuth and elevation angles in degrees. Type `help view` for more information.

Rather than using mesh we can fill in the surface using

```
surf(x, y, z);
```

or using a lighting model

```
surfl(x, y, z); shading interp; colormap(pink)
```

The colour scheme used to shade the surface can be altered by changing the colourmap. This changes the colour associated with a given z value, try

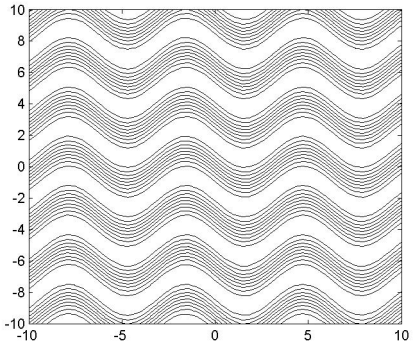
```
>> colormap(gray);
>> colormap(hot);
>> colormap(hsv);
```

Surface plots are attractive for visualisation, but the view chosen tends to distort the presentation of the data. A pseudo-colour plot presents the information as if viewed from above, with z values indicated, not by height, but by colour. Try

```
pcolor(x, y, z);
```

Another approach is to draw curves of constant z-value (these are exactly analogous to contour lines of constant height on a map). Figure 7 shows a contour plot for the function in Figure 6

Figure 7 A contour plot of the function $z = \sin(y + \sin x)$




```
contour(x, y, z, 'k'); % uses all black ('k') contours
```

The contour lines are widely spaced where the function value changes slowly, and closely spaced where it changes quickly (steep surface).