Introduction to Mathematical Modelling for Bioscientists
Workshop 2

author: Kyle Wedgwood
edited by: Daniel Galvis

17th July 2019

Today's objectives

In the second session, we will cover:

- Writing .ode files for *xppaut*
- Model development

1 Writing .ode files

Last time, we learned how to simulate models using *xppaut*. When using *xppaut* we begin by loading previously stored .ode files. These are simple text files that give *xppaut* all the information they need to simulate a system. The syntax used for these files is very easy to understand and requires no prior knowledge about how programming works, making *xppaut* the program of choice for people who don't want to go through the bother of learning to program just to simulate a model.

As a reminder, here is the .ode file for the Brusselator model:

```
# Parameters
par a=1, b=1

# Initial conditions
init x=1, y=1

# ODEs
x' = a + x^2*y - (b+1)*x
y' = b*x - x^2*y

# Memory
@ maxstor=500000
@ bounds=1e7

# Solver options
@ dt=.05, total=60, meth=rk4

# Plotting options
@ xp=t, yp=x, xlo=0, xhi=60, ylo=0, yhi=5

done
```

Let's go through what all these things mean. Anything prefixed by a hash is a comment. These are ignored by the program and are only present to make the code more understandable to humans. As you can see, the code is divided into different sections: parameters, ODEs, memory, solver and plotting options. The order in which these sections appear is unimportant since *xppaut* will parse the entire file and put things in the right place accordingly. This is simply the way that I prefer to order my sections.

Variables Recall that in the Brusselator system, we are interested in the concentrations of the two reactants X and Y. These are our state variables. Each variable in the model should have an equation associated to it. Typically, these are in the form of *ordinary differential equations* or ODEs for short. ODEs describe the *rate of change* of the state variables and will take the form $\dot{x} = \ldots$. Other forms of equation are also possible, but are rarer and we will not consider them here. *xppaut* reads in the ODEs and various parameters and options, then solves these equations to simulate the system.

The Brusselator model is given by the two equations

$$\dot{X} = A + X^2Y - BX - X, \tag{1.1}$$

$$\dot{Y} = BX - X^2Y. \tag{1.2}$$

Compare this to the ODEs in the code to see how this is represented. In *xppaut*, we denote the time derivative $\dot{x}$ by `x'`, where the dash denotes differentiation with respect to time. If you look through the other examples of models we went through last time, you'll see that we can also write `dx/dt`; the two are equivalent. Otherwise, it's essentially 'write what you see'. Many other standard mathematical functions, including sin , cos and all of the other trigonometric functions, are also written in this form.

Parameters In the same system, we assume that A and B are replenished as quickly as they are used. As such, their concentrations never change. They thus cannot be variables. Instead, they appear in the model as parameters. We saw in the last session that by changing the value of the parameters, we can change the behaviour of the system. One of the main goals of mathematical modelling is to understand how the model behaviour changes as parameters vary. The changing of parameters is very often used to reflect interventions into the system, such as the application of a drug.

We can provide parameters to *xppaut* in two ways. In the above code, you will see both of these. The first line sets the value of A to be 1, whilst the second line does the same for B. The difference between the two is that the statement for B is prefixed by `par`. This prefix enables the value of B to modified within *xppaut*, whereas we will not have access to the value of A when running the program. Note that in both cases, you must not leave spaces between the parameter name, the equals sign and the parameter value. Also note that you can set multiple parameter values on the same line.

Initial conditions As well as the ODEs and parameter values, a mathematical model must be provided with initial conditions. In *xppaut*, these can be set as we have done here, by prefixing the allocations `x=1` and `y=1` by `init`, or by writing `x(0)=1` and `y(0)=1`. Both forms are equivalent. If you do not provide initial conditions in your model definition, *xppaut* will set the initial values of all state variables to zero. In most cases, this is fine, but it can occasionally this can cause problems. In any case, it's best to provide initial conditions.

Solving the equation Formally, the correct way to solve ODEs is by integrating them with respect to time. For all but the simplest models, this integration has to be done using numerical procedures, some of which you probably came across in school. There are many numerical routines to numerically integrate models, and *xppaut* has a number of these available to use. Note that none of these methods are wholly accurate; each of them accrues some error as the simulation proceeds. This error is essentially controlled by a parameter δt, which sets how far the simulation advances each time it solves the system of equations. The smaller δt, the more accurate the solution.

They are a number of other options that can be set for all of the numerical methods. Each of these must be prefixed by the `@` sign. As for parameter definitions, you can do multiple of these on the same line. In the `Solver options` section, the first declaration sets δt to 0.05. In this same line, we also set the `total` simulation time to 60, and choose a numerical routine to solve the system. We will discuss this in more detail later.

Memory management In the `Memory options` section, we set a couple of options relating to *xppaut*'s memory usage, again prefixing them by the `@` symbol. The `maxstor` option sets how much system memory is allocated to store the trajectories that are produced when running a simulation. When *xppaut* uses up all of this memory, a dialogue box will pop up informing you of this. As this point, you will either have to stop your simulation, or overwrite the first part of it in order to store new datapoints. This is essentially a remnant of the time when *xppaut* was first written, and memory was at a premium.

These days, it is sufficient to set it to a large number. The same is true of the second option, `bounds`, which sets the maximum value any of your state variables are allowed to take. Generally speaking, if your state variables are reaching the bounds, either something is wrong with your model or your code.

Plotting options The final set of options sets what and how things are plotted in the main window when you first start *xppaut*. The program uses `x` to refer to the x-axis and `y` to refer to the y-axis. This, of course, makes sense, but is also a little confusing for our present example since X and Y are also state variables.

The option `xp=t` chooses to plot time on the x-axis, whilst `yp=x` plots X on the y-axis. The options `xlo` and `xhi` set the plotting range, so that the trajectory between time 0 and time 60 will be plotted in this case. The same is true for `ylo` and `yhi`. If plotting time on the x-axis, it is best practice to set `xlo=0` and `xhi` to whatever value `total` has.

Note that all of these options, initial conditions and any parameter prefixed by `par` are accessible and can be changed from within *xppaut*. Finally, to let the program know the model description is finished, we include the line `done`.

Other options There are a host of other options that can be provided in the code or set within *xppaut* itself. A full description of them can be found in Bard Ermentrout's book[1] or the *xppaut* website.[2] We will cover new options as and when we meet them.

Numerical solvers

In our Brusselator example we set `meth=rk4`. This selects the 4th order Runge–Kutta scheme to integrate our system. This numerical procedure is the textbook standard for simluating ODEs, and so long as δt is set small enough, you won't go far wrong with it. The original Runge–Kutta scheme takes steps of fixed size when it is solving the system.

An improved method is offered by setting `meth=qualrk`. The `qualrk` method still uses the Runge–Kutta framework, but takes timesteps of variable size, aiming to keep the error below a supplied constant. This is set by the option `Toler`, short for *tolerance*. Generally speaking, the default value of `Toler=0.001` should suffice, but if your system is behaving oddly, you can always reduce this.

If your system is such that one or more of your variables changes much faster than the others, the Runge–Kutta solvers may not work well. Using the `rk4` method will result in inaccurate solutions, whilst using the `qualrk` method will be very slow. In this instance, you can set `meth=cvode` instead. This is known as a *stiff* solver and is good for these kind of situations.

Note that at any time in the program, you can change the numerical solver by going into the `nUmerics` submenu, followed by the `Method` option. This wil bring up a list of all of the numerical solvers that *xppaut* supports. Feel free to read their descriptions on the webpage or ask me about them.

Editing the Brusselator model

When we first introduced the Brusselator system, we had parameters that described the rates of the four reactions. As a reminder, the full system of equations was given by:

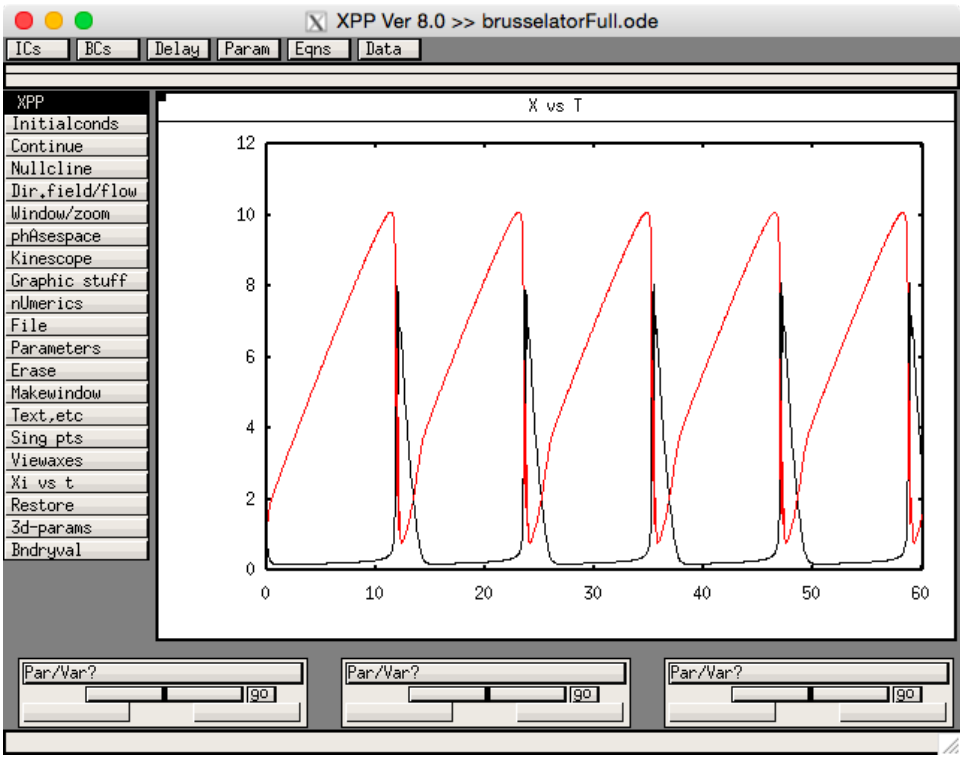
$$\dot{X} = k_1 A + k_2 X^2 Y - k_3 B X - k_4 X, \tag{1.3}$$

$$\dot{Y} = k_3 B X - k_2 X^2 Y. \tag{1.4}$$

Exercise: Copy the original *brusselator.ode* into a new file entitled *brusselatorFull.ode*. Add in the four parameters k_1, k_2, k_3 and k_4 and modify the ODEs accordingly. Start the X server and *xppaut* as before and load the *brusselatorFull.ode*. Experiment by changing the parameter values one by one until you generate the oscillations that the Brusselator model is famous for. After adding the plot for Y to the existing one for X, you should have something that looks like this:

[1] B. Ermentrout. *Simulating, Analyzing, and Animating Dynamical Systems: A Guide to XPPAUT for Researchers and Students*. SIAM, 2002.

[2] URL: http://www.math.pitt.edu/~bard/xpp/xpp.html.



Coding in Matlab

In general, *xppaut* is a great piece of software for rapidly simulating simple models. Since its initial development, other programmers have added more functionality, for example, allowing it to interface with *python*, a programming language which is fast becoming a standard due to its flexibility, usability and the fact that is free. With these additions, and the included *AUTO* package, that we will use next time, *xppaut* can now be used to do most things that you would reasonably want a piece of software for simulating simples models to do.

In spite of *xppaut*'s popularity amongst mathematicians, we cannot get away from the fact that Matlab remains one of the most popular programing languages for scientists. Whilst it is not free, unlike python and *xppaut*, it is easy to use and has many features and libraries included as standard that cover a vast range of different computational areas.

One such area is mathematical modelling. Matlab comes with its own suite of programs to numerically integrate ODEs. These are similar in spirit to the ones we have considered thus far. Now, we will go through one of our previous examples to see how it's written and simulated in Matlab.

For those of you who are familiar with Matlab, you may recall that the language has two types of files: *functions* and *scripts*. The script file is the main driver for any program you write and uses the function files to perform repetitive tasks and to break up the code. It is always good programming practice to keep code in a modular fashion, so that you can edit the different parts separately.

In contrast to *xppaut*, where the entire model was written in one file, in Matlab, we are going to separate this into two, one of which will be a function file, the other being a script. The function file will contain the description of the ODEs only; the parameters, initial conditions, solver and plotting options will all be handled in the script file. Note that, in Matlab, we do not (generally) have to worry about memory options.

The function file To allow for a direct comparison with the *xppaut* code, we will use as an example the Morris–Lecar example of a single neuron. Copy and paste the following into a text editor and save it as *MLeq.m*.

```
function dydt = MLeq(t,y,par)

  % Unpack
  v = y(1);
  w = y(2);

  % Auxiliary functions
  m    = 0.5*(1+tanh((v-par.v1)/par.v2));
  winf = 0.5*(1+tanh((v-par.v3)/par.v4));
  tau  = 1/cosh((v-par.v3)/(2*par.v4));

  % ODEs
  dvdt = 1/par.C*(par.iapp+par.gl*(par.vl-v)...
    +par.gk*w*(par.vk-v)+par.gca*m*(par.vca-v));
  dwdt = par.phi*(winf-w)/tau;

  % Pack
  dydt = [dvdt;dwdt];

end
```

So what do these things all mean? Essentially, these are the same equations that we saw when dealing with the *xppaut* code; just written slightly differently. We are going to take advantage of Matlab's built in numerical integrators. To do this, we have to be a little prescriptive in the way we write our code.

Firstly, we have to specify that this is a function file, which is done simply by starting the code with the keyword `function`. Next, we have to specify an output variable to be fed back to Matlab's solver. Any name will do here, but typical choices are `f` (for function) or `dydt` (to indicate that this is a set of ODEs). We must then put an equals sign, followed by the function name. Note that this should match the filename of the function itself, which is *MLeq* in this case.

In the brackets following the filename, we supply input arguments to the function. For Matlab's solvers, the first argument must always be `t` (for time) and the second must always be a set of state variables (we will come onto this in a minute). After these two, you can supply any number of other input arguments that may be required. Here, we are passing a structure will contains all of our parameter values. There are other, more efficient, ways to pass parameters to our function, but this is probably the easiest to understand from a user's point of view.

Unlike *xppaut*, which stores all of the state variables separately, Matlab's solvers store them in one *vector*. For the uninitiated, this is essentially just an ordered list of numbers. This vector can be given any name you like, but standard practice is to call it `y`. In our case, the first element of this vector corresponds to the cell membrane, V, and the second refers to the fraction of open potassium channels, w. For ease of reading, the first thing we do is unpack `y` into its constituent parts and save these are temporary local variables. Note that Matlab requires to end each line with a semi-colon, else it will display the evaluated value of that line. We really really don't want this to happen, so make sure you include the semi-colon each time you end a line.

Next, we have the auxiliary functions for the Morris–Lecar model. In the *xppaut* code, these are functions of v. In the Matlab code, we already know what the value of v is (since it is being passed in as part of `y`). This means that we can simply evaluate the value of m, w_∞ and τ here. An alternative way would be to write these are their own functions, taking v as an input variable. Note that where parameters are used, they are prefixed by `par.` . This is because the parameter values are all stored in the *struct* `par`. To access them, we need to use the dot notation. In a similar fashion to the state variables, we could also unpack the parameter values into their own temporary variables, but this is not necessary.

After defining the auxiliary functions, we then go on to write the ODEs themselves. In *xppaut*, we could order the sections in any order we wish. In Matlab, however, we must be careful with the order in which we define things. Since the ODEs depend on the auxiliary functions, we must define those first. Note again that parameters must be accessed using the dot notation and also that I have used the ellipsis to write the v equation over two lines.

Like the state variable, Matlab requires the output of this function to be stored as a vector (in this case `dydt`). Specifically, this vector should contain the time derivatives of the state variables in the order that they are listed in `y`. This means that the number of elements in `dydt` should be the same as that in `y`. Matlab will complain at you if you forget to include this step, or if the numbers of elements don't match. Note that `dydt` must be a column vector. We can ensure this by using semi-colons to separate the different entries into it.

No prizes for guessing what `end` does.

We have now defined the system we are going to solve, now we need to set everything else up and simulate it.

The script file Copy and paste the following into a text editor and save it as *simulateML.m*:

```
% Parameters
par.gca  = 4.4;
par.gk   = 8;
par.gl   = 2;
par.v1   = -1.2;
par.v2   = 18;
par.v3   = 2;
par.v4   = 30;
par.vl   = -60;
par.vk   = -84;
par.vca  = 120;
par.phi  = 0.04;
par.iapp = 0.0;
par.C    = 20;

% Initial conditions
v0 = -60;
w0 = 0;
y0 = [v0;w0];

% Simulation options
tstart  = 0;
tfinal  = 1000;
tspan   = [tstart tfinal];
options = odeset('Abstol',1e-8,'RelTol',1e-6);

% Simulate the system
[t,y] = ode45(@MLeq,tspan,y0,options,par);

% Plot the voltage
figure(1);
plot(t,y(:,1),'k');
title('Solution of the Morris-Lecar model');
xlabel('Time (ms)');
ylabel('Voltage (mV)');

% Plot the gating variable
figure(2);
plot(t,y(:,2),'r');
title('Solution of the Morris-Lecar model');
xlabel('Time (ms)');
ylabel('W');

% Save the data
save('MLdat.mat','t','y');

% Save the data in ascii format
X = [t,y];
save('MLdat.dat','X','-ascii');
```

The first section sets the parameter values. Remember that each one needs to be prefixed by `par.` . We could also supply each one separately, but doing things in this way allows us to store all of the parameter values in one place. Matlab doesn't allow us to make multiple parameter declarations in one line, but it does allow us to put whitespace between the parameter name, the equals sign and the value. Putting whitespace in the appropriate place will make the code much easier to read.

After defining the parameters, we then set the initial conditions. Recall that Matlab's solvers expect

the state variables to be given as a single column vector, so we need to construct this. Remember that we must use semi-colons to separate variables in column vectors. Here, the initial state is stored in the vector `y0`.

Next, we need to supply some options so that Matlab's solver knows what to do. The first key thing to provide is the length of time you want to simulate the model for. Here, `tstart` indicates what the time at the beginning of the simulation is. In general, this will be set to 0, but there are situations in which you may want to set this to another value, in particular when you are providing some sort of time dependent input to the model. `tfinal` is the end time of the simulation so that `tfinal-tstart` is the total simulation time.

The main options for simulation via the built-in solvers is set in the `options` variable. This variable is actually the output of a function `odeset`. The Mathworks pages have a full list of options that you can set.[3] Here, we have set the Absolute tolerance and Relative tolerance. The default values of these are generally good enough for most problems, but if your simulation is looking weird, the first strategy to fix it is typically to reduce these two numbers. If you choose not to set any options, you must provide an empty vector using the line:

```
output = [];
```

We are now in a position to simulate the model. The standard Matlab solver is `ode45`, which is a Runge–Kutta method with an adaptive step size, much like the `qualrk` method in *xppaut*. The output of the solver is stored in two variables, `t`, which stores the times at which the solution is computed; and `y`, which contains the value of the state variable at those points.

The first argument to `ode45` is known as a *function handle*. It isn't important to understand the ins and outs of function handles, although you should know that they are an integral part to the way Matlab works, and are very useful. What you do need to know is that the function handle is given by the function name, prefixed by the `@` sign. The second argument is the time span over which you want to simulate the model, which we stored earlier in `tspan`. The third argument is the initial condition, which we stored in `y0`. The fourth argument is the set of `options`. Finally, we provide the `pars` struct that contains all of the parameter values.

At this point, we have simulated the system - the relevant trajectories are stored in `y`. If we want to get a good feeling for what is going on, we need to plot these trajectories. Happily, Matlab comes with a plotting library to facilitate this. We are going to produce two plots: one showing the voltage trajectory, the other showing the w trajectory. Before plotting the data, we have to tell Matlab that we want to produce a figure, which is done using the `figure` command. The number in brackets in a *figure handle*, which is essentially just a label for the figure. The `plot` function actually handles the plotting. The first argument is plotted on the x-axis, whilst the second is plotted on the y-axis. Note that the first column of `y` contains the voltage data, whilst the second column contains the w data. The final argument to the plot function indicates that I want Matlab to draw a blac(k) curve. In the second plot, I instead plot a (r)ed curve. The linestyle page on the Mathworks website tells you all of the options for plotting curves.[4]

The remainder of the commands in the plotting section should be obvious. If you want to save a plot, you simply have to go to `File` → `Save As`. This gives options to save the figure in many different formats. I would strongly recommend saving figures in either .eps or .pdf format. Under the `Edit` → `Edit Axes Properties`, you can change all manner of things associated with the plot to make it beautiful. The default plotting options in Matlab aren't that great, so it's good to familiarise yourself with these options.

Since *simulateML.m* is a script file, all variables are available to us to view and edit from Matlab's workspace. To do, you have only to click on the appropriate variable name in said workspace. We can also save the variables, as shown in the last two sections of the code. The command `save` is used for this purposes. Matlab gives us the option to save data in a variety of formats. The first of these shows how to save variables in a format that Matlab can read, namely as a `.mat` file. The first argument specifies the filename and all subsequent arguments, which must be given in single quotes, are the variables to save. This file can subsequently loaded into Matlab, which will place all of the variables into the workspace.

The second example using the `save` option shows how to save data in a format that can be read by text editors and other programs. Using this option, we first have to put all of the data we wish to save

[3] URL: http://mathworks.com/help/matlab/ref/odeset.html.
[4] URL: mathworks.com/help/matlab/ref/primitiveline-properties.html.

into one variable, in this case `X`. Recall that the data are saved in columns, so we must use a comma to make sure that `X` has the correct form. We use the `save` command again, where the first argument is again the filename, the second is the variable to save and the third is a format, in this case ascii, to save the data. This can now be loaded as a text file in any program that can read data in the ascii format.

Exercise: Convert the original Brusselator model into Matlab code and simulate it using the template code provided above.

A simple biochemical reaction

Suppose that we have a set of chemicals A, B, C and D reacting in a well-mixed container. Assume that A is supplied at a constant rate k_1 and that the chemicals react subject to the following:

$$1) \quad A \xrightarrow{k_2} B \tag{1.5}$$

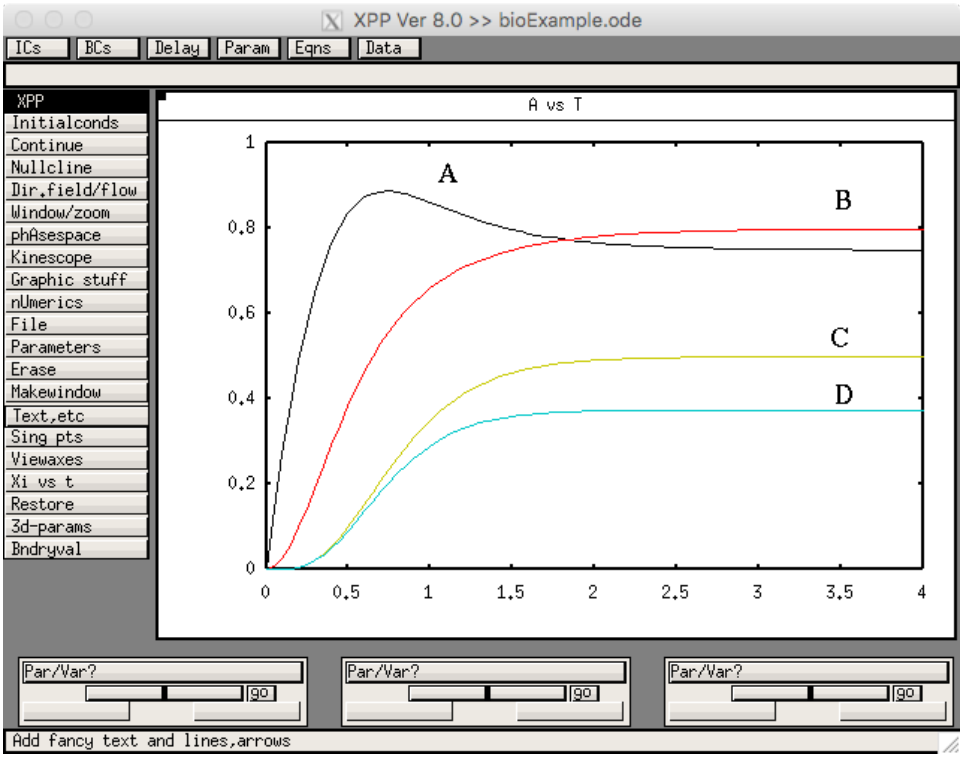
$$2) \quad A + B \xrightarrow{k_3} C + D \tag{1.6}$$

$$\tag{1.7}$$

Finally, suppose that C and D are cleared out of the system at rates k_4 and k_5 respectively, either through conversion to another chemical not involved in our reaction or through decay.

Exercise: Write down a mathematical model for the concentrations of the reactants A, B, C and D. This should be a system of 4 ODEs and should include all of the rates of reaction.

Using your mathematical model, write the system in either *xppaut* or Matlab using the parameter values $k_1 = 3$, $k_2 = 2$, $k_3 = 2.5$, $k_4 = 3$ and $k_5 = 4$. Set the initial concentrations of all chemicals to be zero and simulate the system for 4 seconds. Add curves (in different colours) for all of the reactants. You should have something that looks like this (you can add text labels to your graph using the `Text,etc`→ `Text` menu option):



Determine the steady state concentrations of the species A, B, C and D using *xppaut*. Change the total integration time if needed. Explore the system by modifying the parameters k_1, k_2, k_3, k_4 and k_5. Can you intuitively understand the role that each of these parameters play in the dynamics of the system? Try to produce the same figure as the one above.

Fast and slow processes

Not all chemical reactions take place at the same rate. In fact, some might be significantly quicker than others. Let's explore what this looks like in terms of our system

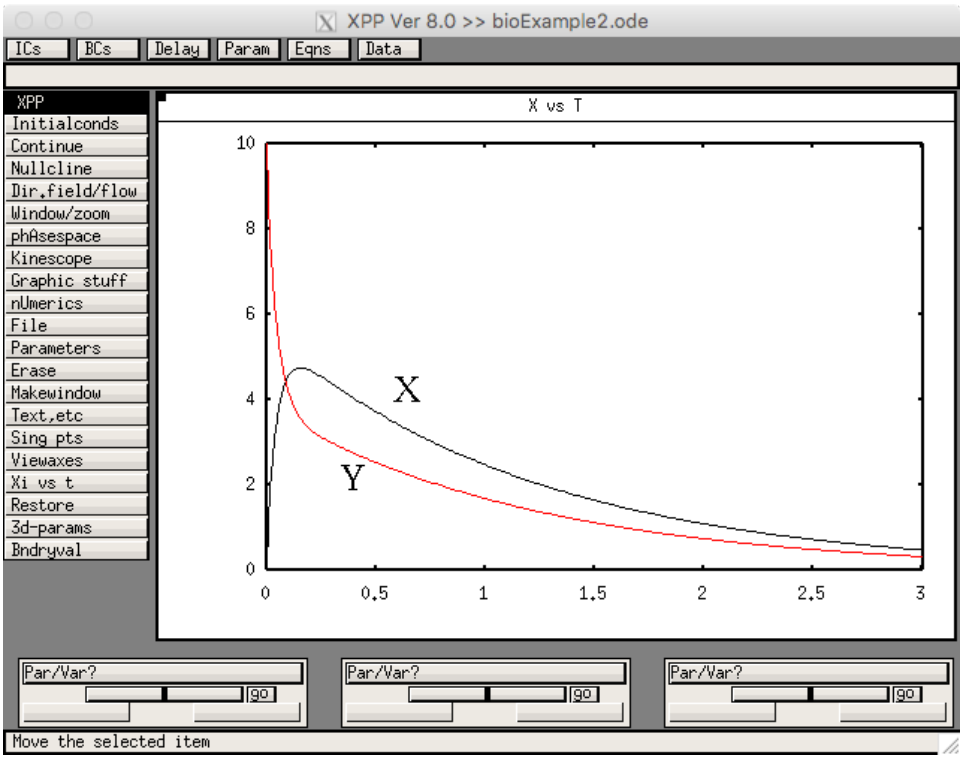
Suppose we have a system of two chemical species, X and Y, which react in a well-mixed environment subject to the following reactions:

$$1) \quad X \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} Y \tag{1.8}$$

$$2) \quad Y \xrightarrow{k_2} D \tag{1.9}$$

$$\tag{1.10}$$

Assume that D is cleared from the system instantaneously. Convert this into a system of ODEs for X and Y and program it in *xppaut* or Matlab. We suppose that the reaction converting X to Y occurs at a much faster rate than the reaction from Y to D. In particular, set $k_1 = 9$, $k_{-1} = 12$ and $k_2 = 2$. Set the initial concentration of Y to be 10 and assume that there is no initial mass of X. Simulate the system for 3 seconds and plot curves for both X and Y to obtain (you may need to lower the time step of the model or use the CVODE solver):



We can see there are essentially two timescales in the model, a fast initial phase, followed by a slower phase. The initial phase is mediated by the fast reaction, as X gets converted to Y. The second phase is essentially controlled by the slower reaction, which removes Y from the system. During this period, the concentration of X is in what we call a quasi-equilibrium. This means that as soon as the concentration of Y changes, X is altered essentially instantaneously. When we analyse mathematical models, we very often take advantage of this observation to reduce the complexity of the model. In this case, for example, we could ignore the equation for X, settings its concentration to be a given function of Y and simulate trajectories for the Y equation only. If you would like to know more about how to do this, please ask me!

Reducing model complexity

We can take advantage of the fast-slow nature of this system by solving the one dimensional system

$$\dot{Z} = -\frac{k_1 k_2}{k_{-1} + k_1} Z.$$

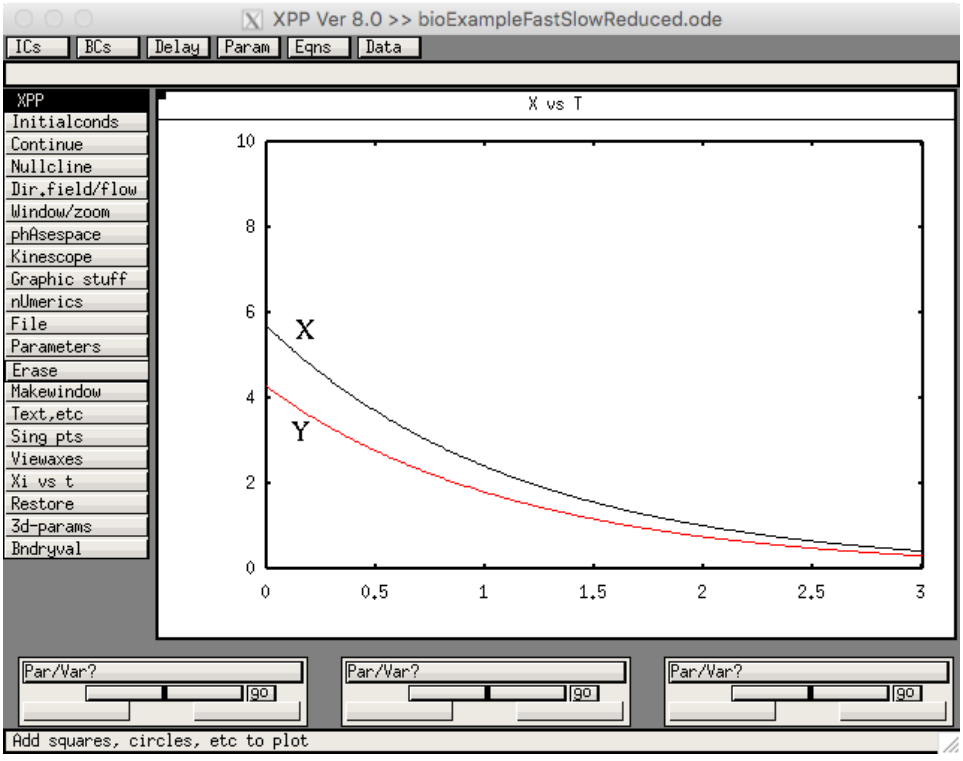
Z represents a pooled variable: $Z(t) = X(t) + Y(t)$, which takes into account the concentrations of both X and Y. Note that this is only an approximation to the original system. The overall characteristics of the model are preserved, but there might be some differences. When performing such an approximation, we are basically interested in the dynamics of the model on either the fast or the slow timescale (in this case the slow timescale). Once we have simulated the trajectory for Z, we can extract those for X and Y using the two formulas;

$$X = \frac{k_{-1}}{k_{-1} + k_1} Z$$
$$Y = \frac{k_1}{k_{-1} + k_1} Z$$

Exercise: Write an *xppaut* file to solve the one dimensional system for Z. Write out the formula for X and Y, prefixing those lines with `aux`, i.e.:

```
aux Y=k1/(km1+k1)*Z
```

Prepending the definition of Y with `aux` tells *xppaut* that these are important auxiliary variables and will allow us to have access to it when we simulate the system. You will now be able to plot X and Y in the usual way (using `Xi vs t` or using the `Graphic stuff` → `Add curve` menu option). After simulating the reduced system, setting the initial concentration of $Z = 10$, you should have:



Compare this to the earlier simulation of the full system for X and Y. What are the key differences? Experiment by modulating the values of k_{-1}, k_1 and k_2. At what point does the reduced system stop being a good approximation to the full system? Can you understand this intuitively given the model description?

From now on, I will introduce models and give instructions of what you should be looking for, but I won't tell you exactly how to get there.

2 Model development

In general, we tend to start from simple models and add in more complexity as necessary. An alternative viewpoint is to start from a complicated model and strip out redundant mechanisms until you end up

with the simplest possible representation of the real world system. However, this process is usually much harder, and you may not have all of the details you need to take this approach. We will now go through some simple model development, based on models we have already encountered.

Accounting for enzymes

So far, we have used a very simple model of biochemical reactions, namely the law of mass action (LMA) Under the LMA, the reactions take a simple, linear form. The 'linear' terminology arises from the notion that if you plot the rate of reaction with respect to the concentration of the reactants, the result is a straight line. In biochemical reactions, the situation is rarely so simple. Many biochemical reactions are catalysed by enzymes, and we should account for these in our modelling descriptions.

Putting together all of the dynamics associated with enzyme dynamics produces a complex, often unwieldy set of equations. Instead of adopting this approach, we can use a much simpler one using approximations arising via time-scale separation. The first and most famous derivation achieving such a reduction was by performed by Leonor Michaelis and Maud Menten. The resulting rate descriptions are known as Michaelis-Menten dynamics.[5]

We will now explore how to account for enzymes in reactions by studying a simple system. Diagrammatically, the system we will study is given by

$$S+E \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} C \xrightarrow{k_2} P + E.$$

In this reaction, S represents the substrate, E is the enzyme, which assists in, but does not get used up during the reaction, C is the enzyme-substrate complex which is formed when the two interact, and P is the product of this relationship. The fact that E exists on both sides of the reaction reflects the fact that it does not get used up during it. The rate k_2 is called the enzyme's *c*atalytic constant.

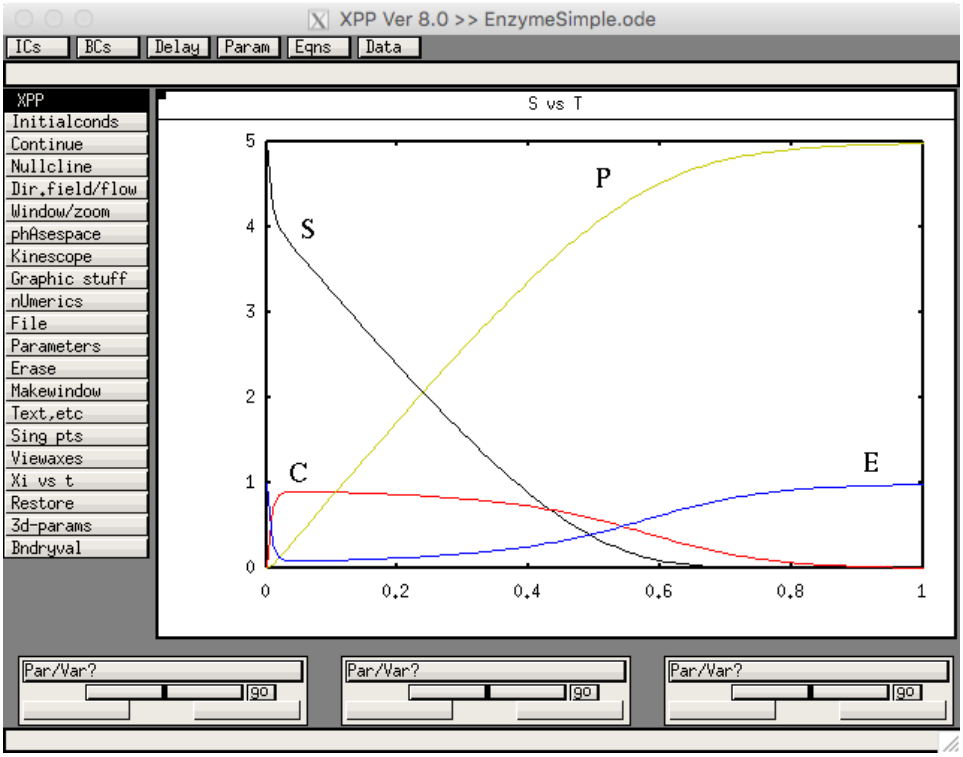
Since the enzyme does not get used up during the reaction, we know that its concentration will remain constant throughout. That is not to say that E will remain constant, but the total amount of the enzyme, if either free or bound form, $E_{\text{total}} = E + C$ will. We can use this conserved quantity to eliminate the dynamics for E.

Exercise: Write down the system of ODEs that govern the simple enzymatic reaction. Using the relationship $E_{\text{total}} = E + C \equiv$ constant, see if you can obtain the reduced system given by

$$\begin{aligned}
\dot{S} &= -k_1 S(E_{\text{total}} - C) + k_{-1} C, \\
\dot{C} &= -k_{-1} C + k_1 S(E_{\text{total}} - C) - k_2 C, \\
\dot{P} &= k_2 C.
\end{aligned}$$

Write code to simulate the reduced system in *xppaut* using parameters $k_{-1} = 1$, $k_1 = 30$, $k_2 = 10$, $E_{\text{total}} = 1$. Use as initial conditions $S = 5$, $C = 0$ and $P = 0$. Set E to be an auxiliary variable, given by $E = E_{\text{total}} - C$. Solve the system over 1 second to obtain:

[5] K. A. Johnson and R. S. Goody. "The Original Michaelis Constant: Translation of the 1913 Michaelis-Menten Paper". In: *Biochemistry* 50.39 (2011), pp. 8264–8269.



In general, the substrate in such a reaction is far more abundant than the enzyme, and consequently the reaction to form the complex is much faster than the reaction by which the product is formed from the complex. Using similar arguments as before, exploiting differences in these natural timescales, we can simplify the whole system down to two simple equations representing one reaction. To do this, we first apply a quasi-steady state approximation for C. This is done by setting $\dot{C} = 0$ and replacing C with C^{qss}. Upon doing this, we get

$$0 = -k_{-1}C^{\text{qss}} + k_1 S(E_{\text{total}} - C^{\text{qss}}) - k_2 C^{\text{qss}}.$$

We can rearrange this equation to find the quasi-steady state for C as

$$C^{\text{qss}} = \frac{k_1 E_{\text{total}} S}{k_{-1} + k_2 + k_1 S}$$

Finally, we substitute this expression back into those for S and P to get:

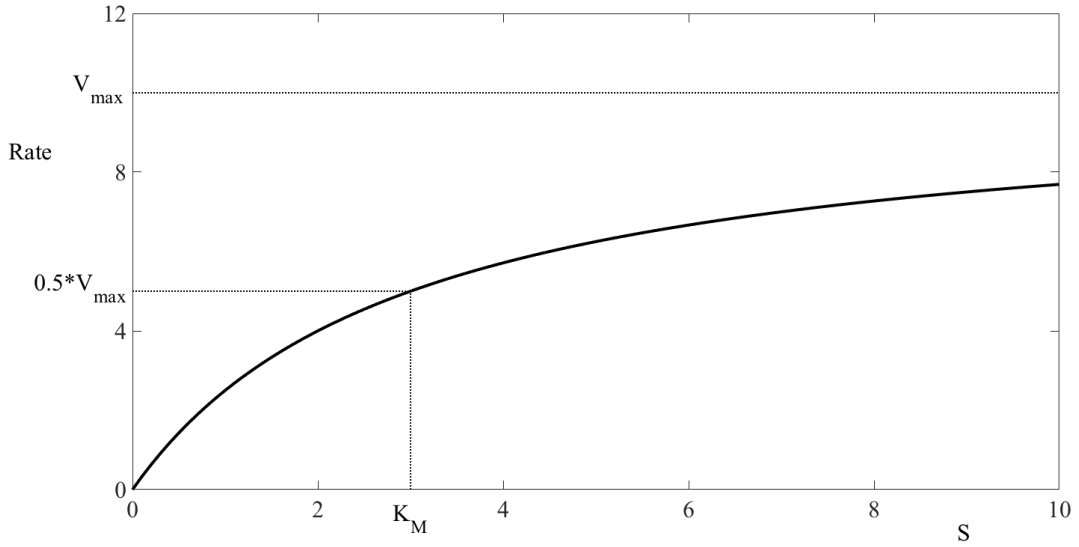
$$\dot{S} = -\frac{k_1 k_2 E_{\text{total}} S}{k_{-1} + k_2 + k_1 S},$$
$$\dot{P} = \frac{k_1 k_2 E_{\text{total}} S}{k_{-1} + k_2 + k_1 S}.$$

This represents the reaction

$$\text{rate of } S \longrightarrow P = \frac{k_1 k_2 E_{\text{total}} S}{k_{-1} + k_2 + k_1 S} = \frac{V_{\max} S}{K_M + S}.$$

Here, $V_{\max}$ represents the maximal rate of the reaction and K_M is known as the Michaelis-Menten constant. It captures the amount of S at which the rate of the reaction is half that of $V_{\max}$.

Exercise: Write code to solve the full system and compare the two to identify any differences between them. Examining the shape of the rate curve, we find that it has the following shape:



We can now clearly observe the nonlinear behaviour of the system. As S gets large, the rate of the entire reaction saturates at $V_{\max}$. When the mass of the substrate is small, however, the rate is approximately linear and so in this situation, the standard LMA will suffice.

Inhibition

Enzymatic reactions rarely occur with only one substrate. Interactions of the enzyme with other substrates can either inhibit or promote the reaction we are interested in. Let's see how to include these in our modelling framework. Consider a system with a competitive inhibitor. We shall denote the concentration of this inhibitor by I. As well as the reactive substrate S, the enzyme E also binds with an inhibitor I to form a non-reactive complex. We shall assume that this reaction is reversible, else we shall quickly run out of enzyme. Denoting the concentration of the non-reactive complex by C_I, we obtain the following (diagrammatic) set of reactions

$$S+E \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} C \xrightarrow{k_2} E + P,$$
$$I+E \underset{k_{-3}}{\overset{k_3}{\rightleftharpoons}} C_I.$$

We assume that the inhibitor is more abundant than the enzyme (in the same way that we do for the substrate). We can thus assume that I remains constant throughout our experiment, so that it enters only as a model parameter. As before, the total amount of enzyme is conserved, so that $E_{\text{total}} = E + C + C_I$ is also constant. Since the amount of both the substrate and the inhibitor are both high, we can assume that the reactions forming both the reactive and non-reactive complex are faster than the reaction generating the end product. As such, we can a quasi-steady state approximation for C and C_I.

Exercise: Write down the equations governing the concentrations C and C_I. By setting $\dot{C} = 0$ and $\dot{C}_I = 0$ in these equations, we find that the quasi-steady state approximation for C and C_I are

$$C^{\text{qss}} = \frac{E_{\text{total}} S}{I K_M / K_I + S + K_M}, \quad C_I^{\text{qss}} = \frac{I(E_{\text{total}} - C^{\text{qss}})}{K_I + I},$$

where

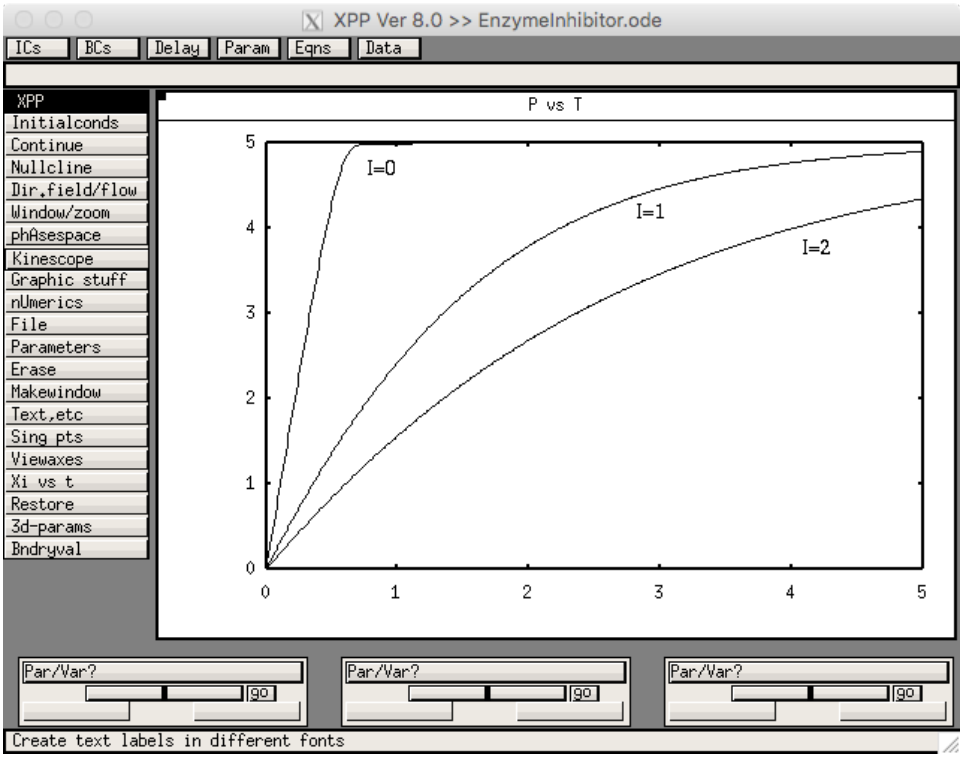
$$K_M = \frac{k_{-1} + k_2}{k_1}, \quad K_I = \frac{k_{-3}}{k_3}.$$

Using these equations, show that the rate of production of P from S can be written using the Michaelis-Menten law (only show for P!)

$$\dot{S} = -\frac{k_2 E_{\text{total}} S}{K_M(1 + I/K_I) + S}. \tag{2.1}$$

$$\dot{P} = \frac{k_2 E_{\text{total}} S}{K_M(1 + I/K_I) + S}. \tag{2.2}$$

Write an *xppaut* code to simulate this model with parameters $k_1 = k_3 = 30$, $k_2 = 10$, $k_{-1} = k_{-3} = 1$ and $E_{\text{total}} = 1$. Simulate the model over a 5 second period, using initial conditions with $S = 5$ and $P = 0$ and vary I to observe how changing the concentration of the inhibitor changes the rate of the reaction. Upon plotting P as a function of time, you should obtain something that looks like this:



Does the final amount of P produced change as you vary I? Can you explain this?

Cooperation

Other substrates can also enhance, as well as inhibit enzymatic reaction rates. This enhancement is termed *cooperativity*, and is best explained using a classic example of oxygen binding to haemoglobin in red blood cells. In contrast to the reaction rate curves we have seen thus far, the rate of oxygen binding in this case follows an S-shape (sigmoidal) curve, much like the activation curves we saw in the Morris–Lecar model. This extra nonlinearity arises from the structure of haemoglobin itself. In fact, haemoglobin has four binding sites for oxygen, and as we will see, this can greatly enhance its efficiency.

In general, let us consider the binding of a molecule X to a protein P:

$$P + X \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} PX.$$

Now, let us consider a protein with many binding sites. We shall denote the fraction of binding sites that are occupied by the variable Y, the *fractional saturation*:

$$Y = \frac{\text{number of occupied binding sites}}{\text{total number of binding sites}} = \frac{[PX]}{[P] + [PX]},$$

where the square brackets denote concentration. The dissociation constant $K = \frac{k_{-1}}{k_1}$ represents a sort of equilibrium constant that tells you how much non-bound P to expect given the reaction we are currently working with. At steady state, we thus have $[PX] = [P][X]/K$. We can then rewrite Y as

$$Y = \frac{[P][X]/K}{[P] + [P][X]/K} = \frac{[X]/K}{1 + [X]/K} = \frac{[X]}{K + [X]}$$

The key contribution of the multiple binding sites comes from the fact that they do not act independently of one another. If this were the case, the overall reaction rate would increase, but we would not observe an S-shaped reaction rate curve. The case for haemoglobin, which possesses four binding sites can be explained by splitting the overall reaction into sub-reactions:

$$\begin{aligned} P + X &\underset{k_{-1}}{\overset{4k_1}{\rightleftharpoons}} PX_1, \\ PX_1 + X &\underset{2k_{-2}}{\overset{3k_2}{\rightleftharpoons}} PX_2, \\ PX_2 + X &\underset{3k_{-3}}{\overset{2k_3}{\rightleftharpoons}} PX_3, \\ PX_3 + X &\underset{4k_{-4}}{\overset{k_4}{\rightleftharpoons}} PX_4, \end{aligned} \tag{2.3}$$

In the above the complex PX_i has i ligand molecules bounds. The rate constants depend on the number of bound ligand molecules. Specifically, the first reaction has rate k_1, the second has rate k_2 and so on. These rates are scaled by stoichiometric factors that account for the number of binding sites involved in each reaction. They essentially represent the number of available binding sites. You should go through these equations to ensure that you understand where these terms come from.

Using our earlier equation, the fractional saturation is given by

$$Y = \frac{\text{number of occupied binding sites}}{\text{total number of binding sites}} = \frac{[PX_1] + 2[PX_2] + 3[PX_3] + 4[PX_4]}{4([PX_1] + [PX_2] + [PX_3] + [PX_4])}.$$

Under equilibrium conditions, we can use the dissociation constants $K_i = k_{-i}/k_i$ for $i = 1, 2, 3, 4$ to write:

$$Y = \frac{[X]/K_1 + 3[X]^2/(K_1K_2) + 3[X]^3/(K_1K_2K_3) + [X]^4/(K_1K_2K_3K_4)}{1 + 4[X]/K_1 + 6[X]^2/(K_1K_2) + 4[X]^3/(K_1K_2K_3) + [X]^4/(K_1K_2K_3K_4)}$$

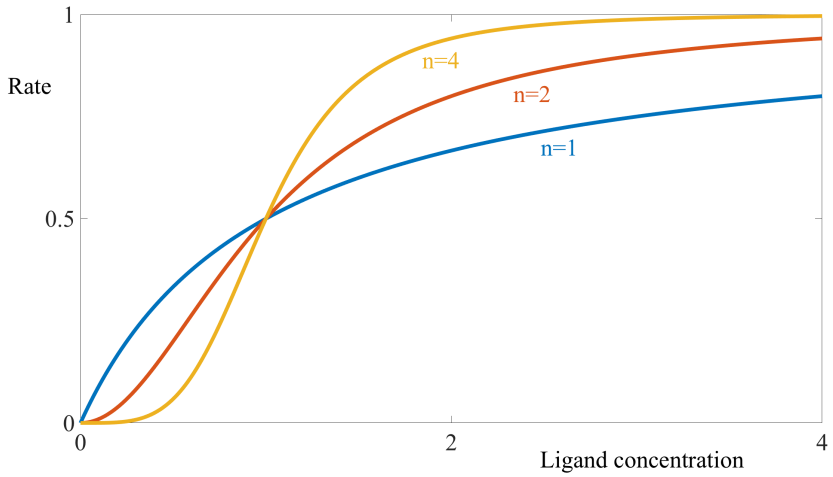
Now, if the later binding events are enhanced by the earlier binding events, we obtain the sigmoidal reaction rate curve. This is known as positive cooperativity. In the extreme case where K_4 is much smaller than K_1, K_2 and K_3, the fractional saturation can be approximated by

$$Y \approx \frac{[X]^4/(K_1K_2K_3K_4)}{1 + [X]^4/(K_1K_2K_3K_4)}.$$

This approximation is formalised in the *Hill function*:

$$Y = \frac{([X]/K)^n}{1 + ([X]/K)^n} = \frac{[X]^n}{K^n + [X]^n}.$$

This approximation was first used in 1910 by the physiologist A. V. Hill as a useful curve for fitting the binding dynamics of haemoglobin. As before, K is the half-saturating concentration of the ligand and can be interpreted as an averaged dissociation constant. The power, n, is called the *Hill coefficient*, and describes the cooperativity of the reaction. Mathematically speaking, it adjusts the steepness of the sigmoidal rate curve: the higher n, the steeper the curve. From a biochemistry point of view, n should be chosen to match the number of events in a multiple binding site process, however, this may not always be well matched to empirical results. Very often, non-integer values are used to provide the best fits to data. One may interpret the sigmoidal curve as acting like a switch, so that beyond a critical value of the ligand, binding occurs at close to the maximal rate. This allows for the separate association and dissociation of the ligand without requiring differences in ligand concentration. Below, you can see how adjusting n changes the rate of reaction.



Exercise: Consider the cooperative reaction given by the reaction

$$P + X \overset{X^n/(K^n+X^n)}{\longrightarrow} PX_n.$$

Write code to solve this system in *xppaut*, setting $K = 1$, using the initial conditions $X = 2$, $P = 10$ and $PX_n = 0$. Simulate the system and produce a figure showing how the rate of reaction varies with n.

Multiple compartments

Up to now, all of our reactions have taken place in what we have referred to as a 'well-mixed' container. In biochemical reactions, this is anything but true. As well as the spatial heterogeneity within the cytosol, there is constant shuttling of many molecules between the nucleus, cytosol, mitochondria, endoplasmic reticulum and other locations within the cell. In addition, molecules may also enter or exit the cell into the extracellular space. We will now discuss the framework for incorporating these effects.

In mathematical terminology, we refer to the differing locations in which the molecules exist as *compartments* and hence the models that include them are referred to as *multi-compartmental models*. These models not only have to account for the reactions including whatever molecules they describe, but also the transportation of these molecules between compartments. Note that, although these models contain more than one location; within each compartment, the reactants are assumed to be well-mixed.

Diffusive processes

The simplest possible way that molecules can move between compartments is through diffusion. Suppose that compartment 1 represents the cytosol and compartment 2 represents the cell nucleus and we have some molecular species S that is present in both. Denote by $[S]_1$ and $[S]_2$ respectively the concentration of S in each of these compartments. The rate of flow of S from compartment 1 to compartment 2 is then given by

$$\text{Rate of flow} = D([S]_1 - [S]_2),$$

where D is the *diffusion coefficient* that quantifies how readily S diffuses across the membrane. This simple form of transport is known as *Fick's law*. In order to capture the relative change in the concentrations of S in each of the compartments, we must account for their respective volumes. In doing so, we obtain

$$[\dot{S}]_1 = -\frac{D([S]_1 - [S]_2)}{V_1},$$
$$[\dot{S}]_2 = \frac{D([S]_1 - [S]_2)}{V_2}.$$

Note that D has units of volume/time.

Exercise: Code this system for simulation in *xppaut*, and set $V_1 = 10$, $V_2 = 1$ and $D = 1$. Choose initial conditions as $[S]_1 = 0$ and $[S]_2 = 1$. Simulate the system over a long time period. What is the resulting steady state concentration? Vary V_1 and V_2 and explore what happens as the relative size of the cytosol to the nucleus changes. How does varying D affect the rate of transport of S between the two compartments?

Curbing infinite grass growth

In the previous session, we encountered the Lotka–Volterra model, that described the interaction of a single prey and a single predator species. As a reminder, this system was given by

$$\dot{x} = \alpha x - \beta xy, \tag{2.4}$$

$$\dot{y} = \gamma xy - \delta y. \tag{2.5}$$

For simplicity, let us again assume that the predators are rabbits and the prey is grass. You may notice here, particularly on the basis of today's session that the first term in these equations represents a linear rate of grass growth. As the population of grass increases, so does its growth rate, even as it gets arbitrarily large. In reality this is not the case, though it may often seem so if you are locked in a constant battle with weeds!

A better representation of a growth process is given by treating the growth rate as a sigmoidal function, much the same as we saw for the cooperative enzyme reactions. This observation has been formalised by the *logistic growth* equation.[6] For now, let us consider just one population, like grass growing on an otherwise deserted island. The equation for the grass population in the absence of predation is given by

$$\dot{x} = \alpha x,$$

In this model, grass grows exponentially with rate α. This rate constant is known as the *Malthusian* parameter, after Thomas Malthus, who performed many initial studies into population growth. In Malthus' original studies, the population simply grows exponentially. Clearly, such exponential growth is unsustainable in the long run. The logistic growth equation, first described by Verhulst in 1845 amends Malthus' equation to take this into account:

$$\dot{x} = \alpha x \left(1 - \frac{x}{K}\right).$$

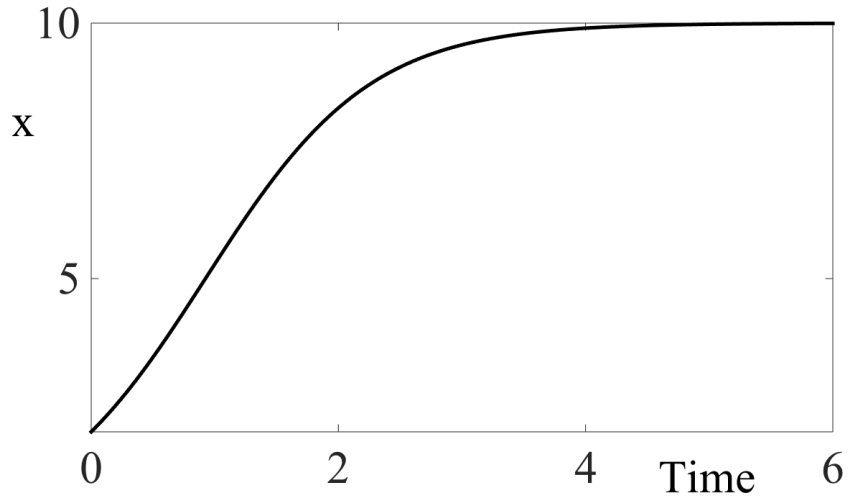
The parameter K is known as the *carrying capacity*. It represents the maximum population size that the environment can support. This value is based on a multitude of things, such as physical space, availability of other resources, accumulation of waste and loss of key nutrients as the population increases. In general, the carrying capacity for a given species in a given environment is impossible to pin down, and may also vary over time as species adapt and the environment itself changes. However, even ballpark figures can be useful in understanding how ecosystems change over time.

Aside: We can solve the logistic growth equation to give the following growth curve:

$$x(t) = \frac{Kx_0 e^{\alpha t}}{K + x_0(e^{\alpha t} - 1)},$$

where x_0 is the initial population size. As time goes on, the population size gets closer and closer to the carrying capacity. As this occurs, the rate of growth gets smaller and smaller as resources get used up. Plotting the the growth curve against time yields a familiar shape

[6] P.-F. Verhulst. "Recherches mathématiques sur la loi d'accroissement de la population". In: *Nouveux mémoires de l'Academie Royale des Sciences et Belles-Lettres de Bruxelles* 18 (1845), pp. 1–41.



This shows that just as normal distributions are everywhere in nature, so too are sigmoids.

Let us now return to the full predator-prey system. Instead of using the original, Malthusian growth rate for the prey species, we can replace it with the logistic growth equation:

$$\dot{x} = \alpha x \left(1 - \frac{x}{K}\right) - \beta xy, \tag{2.6}$$

$$\dot{y} = \gamma xy - \delta y. \tag{2.7}$$

Exercise: Write code to simulate this in *xppaut*. Use the same parameters as for the previous Lotka–Voltera example and set $K = 10$. Simulate the system and observe what happens with the inclusion of the logistic growth term. How is this different from the previous result? As before, plot the predatory population y against the prey population x. Does this help you understand the dynamics of modified Lotka–Volterra model?

We can also include a logistic growth like equation in the predator population. One such approach is offered by Leslie and Gower[7] and is given by the equations

$$\dot{x} = \alpha x \left(1 - \frac{x}{K}\right) - \beta xy, \tag{2.8}$$

$$\dot{y} = \gamma y \left(1 - \delta \frac{y}{x}\right). \tag{2.9}$$

Note that in this model, the carrying capacity of the predatory population is the prey population. This makes sense since we assume that one of the key factors limiting the predator population size is the availability of prey. Making the predator's death rate dependent on the availability of food also makes sense, since the predators should not die from starvation when prey are abundant.

Exercise: Simulate the Leslie–Gower model with $\alpha = 6$, $\beta = 2$, $\gamma = 0.03$, $\delta = 16.7$ and $K = 100$ with initial conditions $x = 25$, $y = 15$. What do you observe?

Lotka-Volterra competition

Lotka–Volterra dynamics do not have to only represent predatory-prey dynamics. They can also represent two non-predatory populations vying for the same environmental resources. Consider the model

$$\dot{x} = \alpha_1 x \left(1 - \frac{x + \gamma_{12} y}{K_1}\right), \tag{2.10}$$

$$\dot{y} = \alpha_2 y \left(1 - \frac{y + \gamma_{21} x}{K_2}\right). \tag{2.11}$$

$$\tag{2.12}$$

[7] P. H. Leslie and J. C. Gower. "The properties of a stochastic model for the predator-prey type of interaction between two species". In: *Biometrika* 47 (1960), pp. 219–234.

Both of the ODEs are logistic growth equations with no predation. However, the competition between the species can lead to interesting effects. In this model, the competitive effects are modelled through the parameters γ_{12} and γ_{21}.

Exercise: Simulate the above system using the parameters $\alpha_1 = 0.2$, $\alpha_2 = 0.5$, $K_1 = K_2 = 100$, $\gamma_{12} = \gamma_{21} = 0.1$ with initial conditions $x = y = 10$. What do you see? Now set $\gamma_{12} = 0.9$ and simulate again. What do you see now? Can you explain this using the model?

Neural bursting

The Morris–Lecar model can represent a lot of nice properties of neural firing, but it can't capture everything. One important behaviour it cannot describe is bursting. Bursting is a neural firing mode in which repetitive action potentials are followed by a long quiescent period. There are a number of reasons why neurons may want to communication via bursting instead of through single action potentials. For example, there is evidence to suggest that neural communication is more reliable when using bursts of action potentials. As is stands, our Morris–Lecar model can only fire action potentials repetitively or remain quiescent, but not a combination of both. In order to describe this behaviour, we need to extend the model to include another (slow) variable.[8]

We will do this by making I_{app} variable, obeying the equation

$$\dot{I}_{\text{app}} = \varepsilon(V_0 - V).$$

Here, ε represents a slow timescale, and must be small. In our simulations, we shall use $\varepsilon = 0.001$ and $V_0 = -22$. For the other parameters, you should use the same values as the Morris–Lecar model we considered earlier, except for V_{Ca}, which you should set to 177 mV.

Exercise: Add the equation for I_{app} to your Morris–Lecar model (it might be a good idea to save this under a new filename). Simulate to system and observe the burst patterns. Vary ε to see how the burst period changes with the model timescales. At what point do the bursts stop? Returning ε to its original value, now decrease V_{Ca} and report how the burst behaviour changes.

[8] E. M. Izhikevich. "Neural excitability, spiking and bursting". In: *International Journal of Bifurcation and Chaos* 10.6 (2000), pp. 1171–1266.