Introduction

Many experiments in learning and decision making require the estimation of **values** and how these change over time. Learning which stimuli and actions have higher value allows humans and other animals to select which stimuli to approach or avoid, and which actions to take.

Some of the most commonly used models of learning and reinforcement were first developed by **Bush and Mosteller** in the 1950s, and further elaborated on by **Rescorla and Wagner** in the 1970s.

They have since developed into the field of **Reinforcement Learning** in computer science.

Our aims for today's session are:

1. get to grips with MATLAB, which we'll be using throughout the four practicals for acquiring, modelling, and analyzing data
2. set up an experimental paradigm of learning and decision making, which you will use to test how model parameters are affected by manipulating subjects' stress levels
3. code a simple reinforcement learning model that can be used to explain behaviour in this task
4. understand how changing parameters in this model affects its behaviour

Parts of the session where you're being asked to do something are indicated with an arrow (→).

Where there are lines printed, use these to make notes about your answer.

Note that the final section (marked ***) is a slightly *harder* section: try and work through this section, and discuss it with the tutors and other students. It's not essential to understand it fully to progress with the later sessions in the block practical. However, if you have understood it, and particularly if you manage to complete exercise D, it will provide a strong foundation for the analyses that you'll do in later practicals.

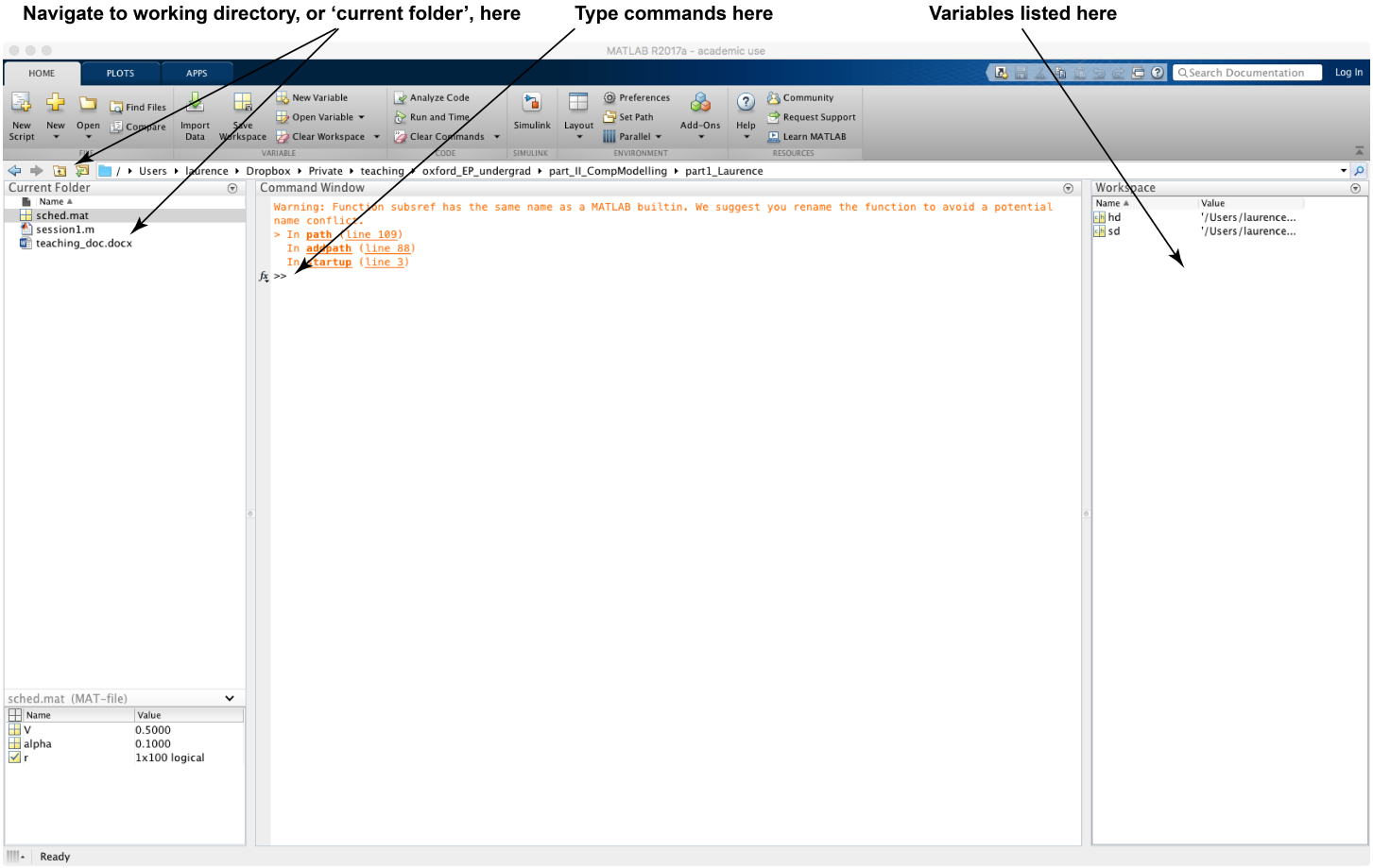
Section 1: Getting started with MATLAB

MATLAB is the currently the most widely-used tool in research for coding, modelling and data analysis. You can use it to acquire data, visualize data, fit models, run simulations, perform statistics, and many other tasks.

More generally, coding is an invaluable skill to learn: it is essential for research, but also important for many careers outside academia. Even if you ultimately end up using a different programming language, such as Python, R, or C++, many of the skills that you learn using MATLAB will transfer extremely quickly.

The best way to get to grips with programming is practice. Try things out, break things, and see what works. If you make a mistake, MATLAB will normally return a helpful error message, telling you what may have gone wrong.

→ Start by opening MATLAB. The exact way to do this will depend on which type of computer you have, but generally if MATLAB is installed there should be an icon you can click. Once opened, you should see something like this:



→ The 'current folder' is the directory where you are currently working. Use the buttons here to navigate to the directory '**Session1_introduction**', where the files for today's practical are located.

Many of you will have encountered MATLAB on previous courses. You may or may not find the exercises in this first section easy – let us know if you want anything to be explained to you.

Check whether you can do each of the things on the list below. If you get stuck during this section, don't worry - move onto the next section after half an hour or so. You'll get more opportunities to learn how to do these things as the practical progresses.

→ Can you do the following?

1. Define two variables, called **a** and **b**, with values 14 and 37. Multiply them together, storing the answer in a variable called **c**.
2. Define a matrix (grid) of numbers, containing 3 rows and 4 columns, called **myMatrix**. Make each element of the matrix equal the product of the row number and column number it belongs to, i.e.

1	2	3	4
2	4	6	8
3	6	9	12

3. Use two (nested) *for* loops to creates a larger version of **myMatrix**, with 25 rows and 40 columns, using a maximum of five lines of code.
4. Use indexing to pick out the *element* of **myMatrix** that tells you what 14*37 equals. Check whether it is equal to the value of **c**.
5. Write a single line of code that takes the 14th row of **myMatrix** and stores it in a new vector called **myFourteenTimesTable**.
6. Use the *plot* command to plot **myFourteenTimesTable**, and the *imagesc* command to visualize **myMatrix**.
7. Write a MATLAB *function* called **reverse_order.m** that takes a vector as its input, and returns the vector in reverse as its output. Use this function to create a variable called **myReverseFourteenTimesTable**

"I'm a psychologist, not a programmer!"

If you're inexperienced in programming, getting started with MATLAB can seem a bit daunting. By far the best way to overcome this is practice. Once you've got used to using MATLAB, things will become much more straightforward. You'll start to appreciate why it's so valuable as a research tool.

A small, bitesize chunk each day is *much* better than trying to cram it all in during a single practical class.

If you're struggling in today's class, we recommend installing MATLAB on your laptop: https://help.it.ox.ac.uk/sls/matlab
And then doing one or two lessons a day from Anatonia Hamilton's 'MATLAB for Psychologists' course online: http://www.antoniahamilton.com/matlab.html

Each of these lessons is quick: 10-15 minutes. You'll be a MATLAB wizard in no time!

Section 2: Running a simple experiment in MATLAB

Now go into the directory called '**decision_task**' by double-clicking on it. This is programmed in MATLAB, using a set of functions called *Psychtoolbox* that has been developed to allow for programming of Psychology experiments in MATLAB (you can download this at http://www.psychtoolbox.org).

In this folder, there are a number of *scripts* and *functions*[1], which are files with the file extension **.m**; the script **decmak_task.m** contains the main program that we'll use to run the experiment in this practical, and it calls the function **run_decmak_trial.m** each time a trial is performed in the experiment. If you want to have a look at what's contained inside these files, you can double-click on them to open them, but please *don't* edit them, as this may cause the experiment to not function correctly.

The aim of the experiment is to study how we learn about which stimuli are likely to be rewarding. We will see how this learning changes depending upon:

(a) the environment you are learning in ('volatile' or 'stable'), and
(b) whether you are being exposed to stressful or neutral sounds.

You are now ready to try the experiment. The experiment should take about half an hour to complete. Complete this in pairs.

→ Practice taking *informed consent* for the experiment: ask your partner to look over the consent information sheet, and then check they are happy to proceed and sign the consent form. Then ask them to read through the task instructions. If you have any questions about the task, please ask one of us.

→ In the command window, type '**decmak_task'** and press enter. It will then ask for *your* initials: type these and press enter.

→ Put on your headphones, type 'S1', 'S2', 'N1' or 'N2' for the condition (as instructed on the slip of paper handed to you), and press enter. Once you have finished the task, the data from the experiment is saved in the folder called 'data'. Open this folder and check that you have been successful in saving the data.

→ You need to collect three more participants on the experiment over the next two weeks. You can do this either on your laptop, or by coming into the Broadbent lab at the times that we have made available. You will need to collect one subject in each of the three conditions that you haven't collected today (for instance, if you completed 'S2' today, then you'll need to test one person on each of 'S1', 'N1' and 'N2').

---

[1] A *script* is a simple program consisting of a sequence of MATLAB commands, that uses and creates variables stored in the MATLAB *workspace*. A *function* is similar in that also consists of a sequence of MATLAB commands, but it is used to perform a transformation of input variables into output variables, and doesn't have access to all the variables in the workspace. Functions can be extremely useful. They are discussed in more details in Lesson 5 of the online 'MATLAB for Psychologists; course, mentioned on the previous page.

Section 3: Coding a simple reinforcement learning model in MATLAB, and understanding the effects of varying learning rates

Answered without doing the task

How did you learn whether green or blue was more likely to be rewarded?

In the stable block, this learning was easier because probabilities stayed constant (75% vs 25%). In the volatile block, they had to adapt faster because probabilities switched frequently (20% ↔ 80%).

How did you weigh this up against the number of points available when making your choices?

Each option displayed a magnitude of reward, which varied independently of the probability of getting a reward. So participants had to consider both:
The likelihood of receiving a reward (learned from experience) The size of the reward (shown explicitly each trial)
They would ideally choose the option with the best expected value, calculated as:
Expected Value=Probability of Reward×Reward Magnitude

How do you think your learning was affected by the sounds that were played over the headphones?

- stress group, unpleasant sounds were intended to induce mild stress. Stress can affect learning by changing how quickly people update their beliefs. For example:
Stress might make participants overreact to recent outcomes, increasing their learning rate.
Stress could also impair learning, making participants less able to track probabilities accurately.
In the control group (neutral sounds), participants' learning was expected to follow the standard pattern: slower in stable blocks and faster in volatile blocks.

Computational modeling tries to describe these questions using mathematical equations. We can then provide a *quantitative* answer to each of them, using *parameters* from the model.

→ Question: what is the difference between a model *parameter* and a *variable*? Write your answer below.

Variable: Changes dynamically during the task or model run. Examples: chosen option, reward received, estimated value of each colour.
Model parameter: A fixed setting that governs how variables are updated. Examples:
Learning rate (α): Determines how quickly estimated values are updated after each trial.
Inverse temperature (β): Determines how consistently the model chooses the highest-valued option.

Learning rate difference across blocks: People are expected to have a higher learning rate in volatile blocks and a lower learning rate in stable blocks.
Stress effects: Comparing stress vs neutral groups could show whether stress changes the learning rate or decision consistency.

Reinforcement learning models provide a way of tracking the probabilities of different outcomes when different actions are taken.

In this task, one of the key problems is to estimate *the probability that green is to be rewarded* on each trial. (Note that this is the *same* as 1 minus the probability that blue will be rewarded – so we only need to track one probability).

We can learn this probability by calculating a **prediction error** on each trial:

$$\delta_t = o_t - p_t \text{ (Equation 1)}$$

where δ_t is the prediction error on trial t, o_t is the outcome (1 if green was rewarded, 0 if blue was rewarded) on that trial, and p_t is the current probability that green will be rewarded (this is called **probOpt1** in the MATLAB code).

We then use this prediction error to **update our expectation** of how likely it is that green will be rewarded in the future.

$$p_{t+1} = p_t + \alpha\delta_t \text{ (Equation 2)}$$

where α is a parameter called the **learning rate**, whose value is $0 < \alpha \leq 1$.

Initialization of the probability; usually starts from chance when it is a learning task.
P(green rewarded)=0.5
This is the most common approach because it reflects ignorance at the start.

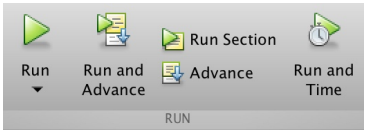
LEARNING RATE :controls how much weight the model gives to new information versus old beliefs.

The learning rate sets the *speed* at which the model learns from previous experience. Let's explore what happens when we vary the learning rate.

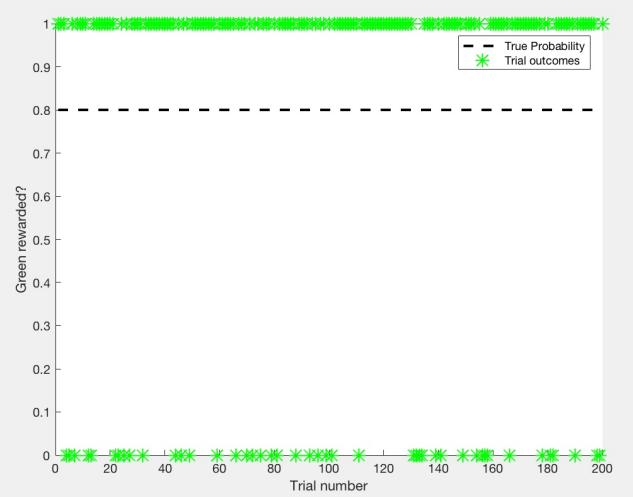
→ Navigate back to the folder named **Session1_introduction**, and open the script called **session1.m**

→ The first 24 lines of code set up, and plot, a simple 'reward schedule' where the *true* probability of green being rewarded is fixed at 0.8. *Read through* these lines of code and try to understand what they are doing. There will be some functions ('rand', 'figure', 'clf', 'hold on') that you might not have encountered before. If you don't know how functions work, type ('help [function name]' in the command window (e.g. 'help figure')). ***The 'help' function is one of the most useful things when learning MATLAB!***

→ Then *run* the code using the '**Run and Advance**' button that can be used to run 'cells' of code:



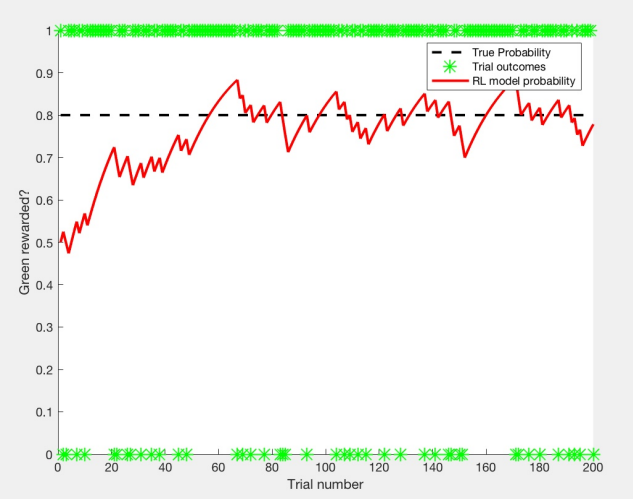
This should produce a plot that looks like this:



The green dots are at 1 every time that green is rewarded, and at 0 every time that blue is rewarded. The black dotted line is the true probability of reward (which the subject doesn't know in the experiment).

→ Exercise A: The function **RL_model.m** (called in line 27) takes as its input: whether green is rewarded on each trial (**opt1rewarded**), what the model's learning rate α is set to, and what the starting probability on the first trial is (p_1). It tries to return **probOpt1**, the probability of green being rewarded, as its output. However, the final two equations in the function haven't been completed. *Open **RL_model.m** and complete the final two lines of code.*

Once you've done this, you should be able to run this cell without receiving any errors, and the figure should now have a red trace that approaches the true probability:



→ Exercise B: Now try playing around with the four parameters at the top of the cell: **fixedProb, startingProb, alpha** and **nTrials**. Particularly try to understand the effects of varying **alpha**. What are the advantages of having a low α? What are the advantages of having a high α? If you set α to 1, how does the model behave?

✅ Low α (e.g. 0.05)
SMALL UPDATES each trial.
The model is slow and steady: it doesn't overreact to single outcomes.
Advantage: good when the world is stable (true probability doesn't change). The model gradually converges to the correct probability.
Disadvantage: very slow to adapt if the environment suddenly changes.

✅ High α (e.g. 0.9)
Large updates each trial.
The model REACTS strongly to NEW outcomes.
Advantage: good when the environment is volatile (probabilities change often) — the model adapts quickly.
Disadvantage: predictions become noisy, swinging up and down depending on the last few outcomes.

✅ α=1
The model completely FORGETS the PAST and bases its belief only on the ONLY MOST RECENT outcome COUNTS.
If rewarded → prediction jumps straight to 1.
If not rewarded → prediction jumps straight to 0.

In the experiment you performed in section 2, the reward probability isn't fixed, but it *reverses* at various points during the experiment – so at some points green is more likely to be rewarded, but at other points blue is more likely to be rewarded.

Let's now change the script so that we load in a schedule such reversals take place.

→ In line 12 of the code, carefully type the following:

```
D = load('schedule.mat');
trueProbability = D.trueProbabilityStored;
nTrials = length(trueProbability);
```

This overwrites **trueProbability** with a schedule that reverses, like in the experiment that you performed. Now try re-running this cell, to see whether your reinforcement learner can keep track of this changing probability.

The learning experiment that you performed in section B has **two manipulations** built into it:

i. Different subjects are exposed to either stressful or neutral sounds. We'll use this to compare whether this affects peoples' learning in the experiment.
ii. There are periods where the reward environment is *stable* (it doesn't change very frequently) and other periods where the reward environment is *volatile* (it reverses quite frequently).

We'll return to the stress manipulation in later sessions. With regard to point (ii), our hypothesis is that people *adjust their learning rates* depending upon the stability of the environment.

→ Exercise C: Let's assume that subjects want to get their estimated probability as close to the *true* probability as possible. Based upon what you learnt about varying α in Exercise B, in which environment might it be helpful to have a lower α? In which environment might it be helpful to have a higher α?

Since the value of a depends on the previous trials (how deep in the past we look for learning and a low a = 0.1 means it is stable, not many changes, we see in the past it is better for a stable environment, while higher a is better for a volatile environment.

The reasoning behind why you might need to have a different α in different situations is discussed more fully in the following paper:

Behrens, T. E. J., Woolrich, M. W., Walton, M. E., & Rushworth, M. F. S. (2007). Learning the value of information in an uncertain world, *Nature Neuroscience 10*(9), 1214–1221. http://doi.org/10.1038/nn1954

α is a learning rate parameter, between 0 and 1, which determines the size of the update step. Its interpretation is clearer in an algebraically rearranged form of the update rule, Vk+1(Sk)=(1-a)Vk (Sk) +a*rk. This form reveals that the error-driven update accomplishes a weighted average between the observed reward (with weight α) and the previous reward prediction (with weight (1-α)). Thus a larger learning rate updates the value prediction to look more like the current reward and a smaller learning rate relies more on older estimates than on the current reward.
Vk(sk) = your current estimate of value for state sk at trial k
"Value" = how good or rewarding you expect this state to be
sk is the state or situation you're in on trial k
rk = the reward you actually received on trial k

*** The key insight here is that the *learning rate, α*, determines how the currently estimated probability on the next trial, p_{t+1}, is influenced by the past history of trials. We can think about this in another way. Look back at equations 1 and 2. p_{t+1} depends upon the outcome of the most recent trial, o_t, but also the last trial's probability estimate, p_t. However, p_t could itself be written in terms of the probability of the previous trial's outcome, o_{t-1}, and p_{t-1}. In turn p_{t-1} could be written in terms of o_{t-2} and p_{t-2}. And so on.

δt = Ot - pt (Equation 1)
pt+1= pt + a*δt (Equation 2)

In short, it becomes possible, when you are at trial T, to think of p_{T+1} as a *weighted sum* of all the previous outcomes that the model experienced:

$$p_{T+1} = (1-\propto)^T p_1 + \sum_{t=1}^{T} w_t o_t \text{ (Equation 3)}$$

Equation 3: "Current estimate = weighted sum of all past outcomes + starting guess"
Equation 4: "Weight for each outcome decays the further back it is"

Where p_1 is the starting probability (this becomes less and less important as we get further away from it, as $(1-\propto)^T$ shrinks to zero), and w_t is the amount of *weight* given to the outcome on trial t on the current trial:

T–t = how many trials ago it happened

tells you exactly how much influence each past outcome has on your current estimate

$$w_t = \propto (1-\propto)^{T-t} \text{ (Equation 4)}$$

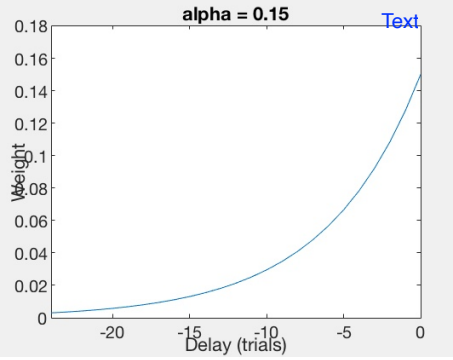
Most recent trial: T–t=0⇒wT=α → biggest weight
Older trials: as $T-t$ increases, (1–α)T–t shrinks → weight fades exponentially

T is the current trial number, and so (T-t) indexes how many trials into the past we are looking. In the final exercise, we see if you can *derive* this equation, step-by-step.

α decides the speed of updating and how much past trials are discounted.

What does w_t actually look like? This is explored in the last cell of **session1.m**, which plots equation 4:

Computes a weighted running average of all rewards received previously in the presence of the stimulus, with - the most recent reward weighted most heavily and -the weight for prior rewards declining exponentially in their lag. Here, the learning rate can be equivalently seen as controlling the steepness of the decay,
with higher learning rates producing averages more sharply weighted toward the most recent rewards.
Such an exponential pattern is a key hallmark of this sort of error-driven updating.



The most recent outcome gets the biggest weight.
Older outcomes fade away exponentially.

The starting probability
$p0$ becomes almost irrelevant after many trials.

This plot is also discussed at the beginning of the following book chapter (which is available as a PDF online at http://www.princeton.edu/~ndaw/dt.pdf):
Daw, N. D. and Tobler, P. N. (2014) Value learning through Reinforcement: The Basics of Dopamine and Reinforcement Learning.
Chapter 15, Neuroeconomics (2nd edition), edited by Glimcher, P. W. and Fehr, E.

→ Try varying alpha, replotting the weights using the final cell of the script, and seeing how it affects the weight assigned to the history of outcomes.

→ *** Exercise D. Last but not least, a bit of maths. Starting with equations 1 and 2, see if you can *derive* equation 4, by substituting terms from the previous trial's equation for p_t. (This is the hardest exercise we've asked you to do today. If you manage to complete it, well done!).