NST Part II GENETICS

Lent Term **Matlab Practical Class**

Genetics – John Welch

This practical class will use Matlab, a powerful numerical programming environment that can be used to do calculations and to analyse and present data.

During the practical you will write a computer simulation of evolution, which follows allele frequencies changing over many generations. The simulation will be based on the experimental evolution study of Buri (1956; Evolution 10: 367-402), which demonstrated the effects of genetic drift.

If you have some experience of Matlab, or another programming language, then some of earlier stages of this practical may be unnecessary, and in later stages, you may discover alternative approaches and shortcuts that may allow faster solution of the problems presented. Feel free to explore these, but here we will present the method that we think is easiest to understand.

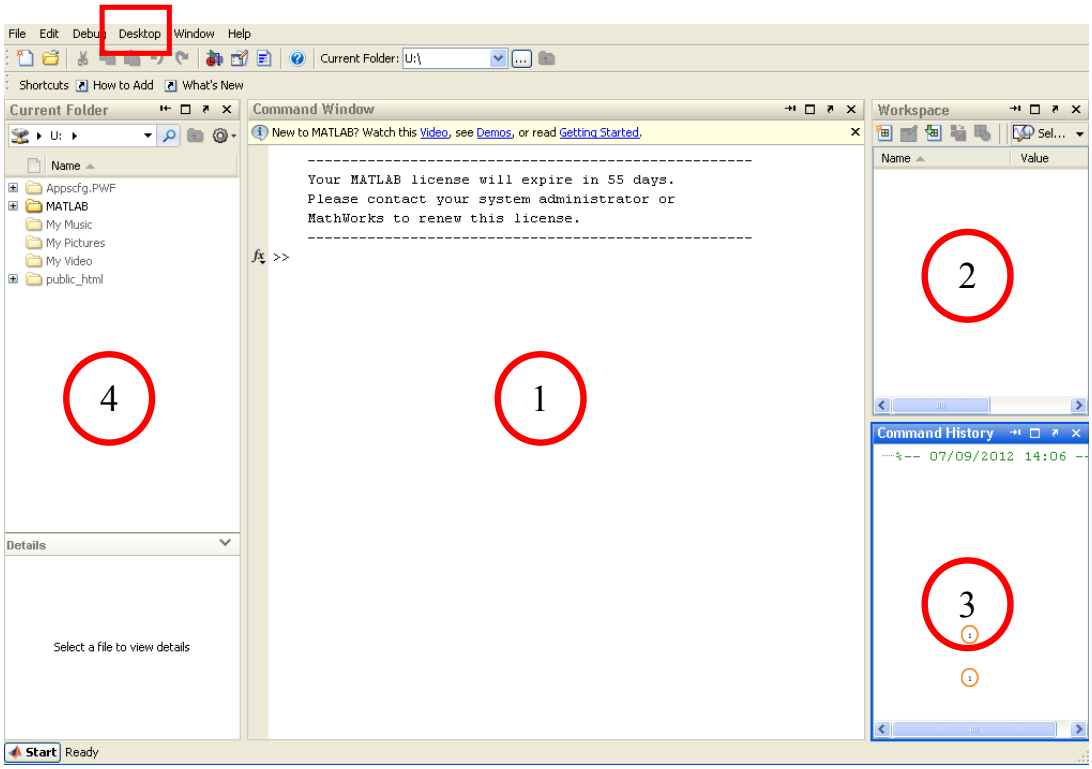
If you have never tried any computer programming before, then you might find that you do not have time to reach the later stages of the practical. **Do not worry if you do not have time to complete the worksheet**. The biology covered here reiterates material covered in lectures, and our major goal is to give you some hands-on experience of programming.

Getting started

If you have not used Matlab before please take some time to read through the following introduction, which will enable you to get the most out of this practical.

First, start up Matlab by clicking its icon on the taskbar. It looks like this: .
Note that Matlab is also accessible from your college computer rooms by logging on to the PWF – it can be found by clicking on the **Start** Menu, then **All Programs > Spreadsheets Maths and Statistics > Matlab**.

The main components of the Matlab working environment are shown in the image below.



If for some reason your layout is different from the image above, find **Desktop** (boxed in image above) on the toolbar menu along the top (where **File, Edit,…** are located) and select **Desktop Layout > Default**. This will return the Matlab environment to its default layout.

Please take a little time to have a look at the layout of the screen. The components of interest are as follows.

1. **Command Window**. This is where you interact with Matlab. Commands are typed here. This is also where results are printed.
2. **Workspace**. Here all the variables you create during a Matlab session are shown. You can double-click on a variable here to display its value.
3. **Command History**. This shows all the commands that have been typed into the Command Window (so it should currently be empty). You can re-execute any command from the Command History by double clicking the command or dragging the command to the Command Window with the mouse. You can also use the up and down direction keys while in the Command Window to find previous commands.
4. **Current Folder**. Here you can see your current working folder. By default this is located in **U:** (your personal folder on the MCS). You can navigate the file system from this window.

Don't worry if you do not understand what all these components do at the moment. It will become clearer as you start using Matlab.

So let's start using Matlab. Anything that you are required to type will be in `Courier New` font. Typically it will be boxed too. When you see the >> sign, this is an indication that the text that follows should be typed into the Command Window. **DO NOT TYPE '>>' when entering these commands.**

1. Using Matlab as an elaborate calculator

Type the following into the **Command Window.** Press the **<Enter>** key on your keyboard after each line of text and look at what output Matlab gives. You do not need to type in the text after the %.

```
>> 2.25 + 3.01
>> pi       % Matlab ignores anything after the percent sign
>> 2.* pi   % You don't need to type these comments
>> 2.^3     % Sometimes we use them to explain what you are typing
```

WARNING. Note from the third and fourth lines above that in Matlab the symbols for multiplying, dividing and taking powers have a '**.**' before them. These operators written without a dot have different meanings. Addition and Subtraction never have a dot before them.

Add	Subtract	Multiply	Divide	Power
+	-	.*	./	.^

(Matlab follows the laws of 'Order of Operations' i.e. brackets take precedence over exponentiation which takes precedence over division which takes precedence over multiplication and so on.)

Tip: You can suppress the output of a command by using a semi-colon ';' at the end of a command.

2. Built-in functions

Functions are commands that require one or more inputs and produce one or more outputs. Matlab has thousands of built-in functions. To find out more about any particular function you can use the `doc` command. To test this type:

```
>> doc round
```

Notice Matlab also suggests similar functions that may be of use.
Many functions can take more than one input. These are separated by a comma. For example:

```
>> round(1.51)     % rounds the input to nearest integer
>> min(2,4)        % returns the smaller of the two numbers
```

You can explore the wide range of functions Matlab has by clicking on the  button. It can be found near the top of the Matlab console.

3. Variables

Variables are one of the most important concepts in computer programming. The text below shows you how to set up variables named "r" and "area" and use them in another calculation:

```
>> r = 2.5                % Set up a variable "r" and set its value to 2.5
>> area = pi .* r .* r    % set up "area" and use "r" to set its value
>> r = 2                  % Change the value of "r" from 2.5 to 2.
>> area                   % Check if this has changed "area"
```

Note that the value of `area` has not changed. It is important to realise that, unlike for example Microsoft Excel, Matlab stores only the value of `area` at the time it was defined, not the equation defining it.

Variable names must start with a letter and can be any sequence of letters, numbers and underscores ('_'). Variable names are case-sensitive so the variable `area` is not the same as the variable `Area` (try typing `Area` into the command window).

Tip: you can call your variables anything you like – it is often helpful to give them names that help you remember what they were for e.g. "generation", "time", "days" etc.

Vectors

So far we have created variables containing a single number, i.e. a "scalar". We often need to store a list of values under one variable name and may want to perform the same calculation on each value in the list e.g. add 1 to each value. For this we use a "vector", i.e. a list of scalars. There are a number of ways to do this in Matlab:

(a) *Creating a vector from a list of values*

```
>> vector1 = [1,1,2,3,5]
>> vector2 = [8 7 2 9 5] % note that the commas are optional
```

(b) *Creating a vector whose entries all have the same value*

When you want many entries of a vector to be the same, this can be done automatically. This is easiest for vectors of zeros and ones.

```
>> zeros(4,1)     % creates a vector of four zeros.
>> zeros(4)       % Note that without the second argument, "1", we get a 4x4 matrix
>> ones(16,1)     % creates a vector of 16 ones.
>> fives = 5.*ones(6,1) % define the variable "fives" as a vector of six 5s.
```

(c) *Creating a sequence of numbers*

To create a vector containing equally separated values, a *sequence*, you can use colons (':'). The syntax to make a sequence is

```
variable = start:stepsize:end
```

```
>> one2five = 1:5                    % the default stepsize is 1
>> smallerincrement = 1:0.5:9        % what values does this generate?
>> godown = 10:-2:0
```

(d) *Creating a vector of random numbers*

There are lots of built-in functions for creating vectors of random numbers. For the practical we will only need random integers (whole numbers)

```
>> randi(10,3,1)       % outputs three random integers between 1 and 10
>> randi([5 10],3,1)   % outputs three random integers between 5 and 10
```

(e) *Creating a vector from external data*

Finally, it is also possible to create a vector in a spreadsheet or to import data from another source. We will not explore this in this practical but it may be useful for you to explore in your own time if you want to use Matlab to help you present your experimental data: for details click on the  button and click on the links: Matlab>Demos>Getting Started>Importing Data from Files).

4. Manipulating vectors

4.1 Selecting/changing an element in a vector

If you have a vector stored in a variable, for example `MyVector`, you can pick out particular elements by using `MyVector(n)` where `n` is the position. The index `n` can also be a vector. In this case the elements of the vector specify which parts of the original `vector` are to be extracted.

```
>> MyVector = 1:3:15
>> MyVector(3)              % what output does this give?
>> MyVector(3) = 182;       % redefine the value of the vector at position 3
>> MyVector
>> MyVector(1:3)            % display the first three values in the vector
>> MyNewVector = MyVector   % creates a new vector whose value match the old one
```

4.2 Performing calculations on vectors

All the usual mathematical operators and many of Matlab's built-in functions can be directly applied to vectors. Remember that it is important to include the "dot" before the multiply, divide and power signs (otherwise Matlab will try to perform the operations using Matrix algebra methods and may give you the wrong answer or produce an error message).

```
>> MyVector = 1:9
>> MyVector.*10
>> MyVector.*MyVector
```

5. Scripts

Entering commands into the **Command Window** and executing them one at a time is tedious for long sequences of commands, or if you want to repeat the same command but with different numerical values. These problems can be avoided by using scripts. Scripts are simply files containing a sequence of Matlab commands. Running a script in Matlab will execute the sequence of commands in the script as if they were typed one after another into the **Command Window**.

5.1 Creating a script file

To create a new script go to **File > New > Script**. This should open a blank sheet in the **Editor Window**.

Check your understanding:
In the blank script file type the commands necessary to

(a) Create a vector, `t`, of numbers from `0` to `5` with an increment of `0.01`.
(b) Create another vector, `y`, which is defined as `y=exp(t)`
(c) Plot the function corresponding to `y` using `plot(t,y)`

Save the file as **plot_exp** in the default folder (Click **File > Save** or click on  in the editor window). Matlab will automatically give this file a '.m' extension.

5.2 Executing (= running) a script file

There are at least three ways to execute your script (i) type `>> plot_exp` in the Command Window; (ii) Press CTRL-ENTER when you are in the editor; or (iii) Click on  in the editor window.

Whichever way you choose to use, once you execute your script a new window should appear entitled Figure 1 with a graph in it. This figure can be 'docked' to your Matlab console by clicking on  in the top right corner of the figure window. You can hide any windows in the console you aren't using by clicking on  . You can also drag the windows around to whatever format suits you. Remember you can always get back to the default window settings by selecting **Desktop > Desktop Layout > Default**.

WARNING! Variables defined in script files become available to use in the command window. This means it is possible to overwrite variables and built-in functions through script files.

6. Relational operators and conditionals

To extend Matlab functionality beyond simple equations and formulae we will need to make use of relational operators and conditionals. These allow us to make decisions in our programs based on the properties of variables.

6.1 Relational operators

Relational operators take two arguments and return a *true* or *false* value. In Matlab *true* and *false* are equal to 1 and 0 respectively.

```
>> a = 1; b = 3;
>> a == b % tests to see if a equals b
>> a ~=b  % tests to see if a does not equal b
>> a > b  % tests to see if a is bigger than b
```

Sometimes we also need to test whether or not multiple relations are true

```
>> a = 1; b = 3; c = 4;
>> a == b | a < c % true if a equals b or a is less than c
>> a ~=b  && b < c % true if a not equal to b and b is less than c
```

6.2 Conditional statements: `if`, `else` and `elseif`

Conditional statements enable you to control how your code will run. Type the following examples into your script file to explore the main ways conditional statements can be used. Before you press CTRL-ENTER think about what you expect the output to be.

`If`, `elseif`, `else` statement

```
%% if, elseif, else statement
a = 5;
b = 5;

if a < b
      disp('a is less than b')
elseif a==b                          % Check another condition if the
      disp('a is equal to b')        % if statement is false
else
      disp('a is greater than b')
end
```

The `elseif` and `else` statements are optional and you can have multiple `elseif` statements.
Check if code above behaves as you would expect when you change the values of `a` and `b`.

Note on indenting: Indenting is not necessary to make the program run; however, it does make the code clearer. Matlab will automatically try to indent your code when you type if statements. If the indenting gets messy as you add bits into your code you can use CTRL-I when in the script window to "smart indent" your code – i.e. Matlab works out how it should be indented and does it for you.

7. Loops – this section is very important!

Many calculations we perform require us to perform simple loops. Loops allow us to repeat sections of code. The best way to learn and understand what is going on is to try out bits of code – type in each of the examples below into a script file and make sure the output makes sense to you:

```
% introduction to for loops
% Example 1: print something 5 times
for i = 1:5
    disp('XXX is the best.') % Insert your own name here.
end

% Example 2: output the value of iterator
for i = 1:5
    i
end

% Example 3: find the square of the iterator
for i = 1:5
    i.^2
end

% Example 4: Create a vector of length 5 that takes the values 1 to 5
clear all      % This command wipes clear all of the variables defined previously
for i = 1:5;
    n(i) = i   % Note that the complete vector is printed every time.
end
```

```
% Example 5: create a sequence so each value is some function of the
% previous value in the vector
clear all
n(1)=2
for i = 1:4
    n(i+1) = n(i).*n(i)
end       % n should end being a five-element long vector

% Example 6: increase age by 12 months each year
clear all
n(1)=0
for i = 1:4
    n(i+1) = n(i)+12
end
```

8. Using Matlab to simulate Buri's (1956) genetic drift experiments

Buri evolved (small) populations of $N = 16$ *Drosophila melanogaster* in the lab. At the beginning of the experiment, a proportion of the flies were homozygous for a "bw75" allele, which alters eye colour, but does not affect the fitness of the flies. The other flies were homozygous for the alternative wild-type allele. Each generation, Buri allowed the flies to mate, and kept N surviving offspring. These N flies then became the parents for the next generation. By examining their eyes, Buri could calculate the proportion of the alleles in the population that were bw75 in each new generation. Buri was particularly interested in the number of generations that elapsed before one or other allele reached fixation.

We will use Matlab to simulate Buri's experiments, but with some important simplifications. First, we will treat the individuals as haploid, so that there are N alleles present in each generation (rather than $2N$ with Buri's diploid flies). Second, we will assume that there are no sex differences (our individuals will be hermaphrodites). Third, we will assume that no natural selection is acting at all (Buri could not ensure, for example, that deleterious mutations did not arise at other loci in his flies' genomes).

Given these assumptions, let us write down the steps we need to follow to calculate the allele frequency in each generation to help work out how to structure the code:

Step 1: Set any parameter values
- Population size (N)
- Proportion of population in generation 1 with the bw75 allele
- Number of generations that the experiment will last

Step 2: Set up the initial population
- Assign an allele type to each member of the parent generation

Step 3: Create a vector for the allele types of offspring
- Choose N parents at random to contribute to the next generation. Note that these parents should be chosen *with replacement*, so that any parent could, by chance, contribute more than one offspring.
- Copy the N alleles of randomly selected parents to form the offspring generation
- Record the proportion of bw75 alleles in the offspring.

Step 4: Go back to step 3 taking the parent generation to be the offspring generation created in step 3. Repeat this process for the number of generations set in step 1.

Step 5: Plot results (i.e., plot the frequency of bw75 over time)

Step 6: report number of generations required to reach fixation

There are a number of ways you can write the Matlab code from the above "pseudo-code". Given a population of 16 individuals with the bw75 initially present at frequency 0.5, have a go at writing the code yourself (although a solution is provided below if you get stuck).

NEED HELP? ASK A DEMONSTRATOR!

Tip1: It will be is easiest to represent the population as a vector, where the wild-type allele is represented by a zero, and the bw75 allele by a one.

Tip2: Because step 3 is repeated a fixed number of times, it should appear in the middle of a `for` loop.

Tip3: Step 6 might be difficult, and there are lots of different ways to do it (some much better than the solution shown in the our example code).

```
% clear workspace and any figure windows
clear all
clf
% Step 1
N = 16; % population size
NumGenerations = 500;     % stop the expt. after this many generations.
InitialFreq = 0.5;        % The initial frequency of the bw75 allele

% Step 2: Set up the initial population
Numbw75 = round(N.*InitialFreq); % How many individuals carry the bw75 allele?
Parents = zeros(N,1); % Create an N-length vector of zeros
Parents(1:Numbw75)=1; % Change some of the initial alleles to 1, representing bw75

Propbw75(1,1) = sum(Parents)./N; % Record the initial frequency of the bw75 allele

% Set up the loop that will contain "Step 3" (Note that it starts at generation 2)
for generation = 2:NumGenerations

  % Let us know which generation we are currently on
  display(['Creating the offspring for generation',num2str(generation)])

  % Step 3: Choose "N" parents at random (i.e., N random integers between 1 and N)
  LuckyParents = randi(N,N,1);
  % Create the offspring generation
  Offspring = Parents(LuckyParents);

  % Record the frequency of the bw75 allele in this generation
  Propbw75(generation,1) = sum(Offspring)./N;
  % The children grow up, and become the next generation of parents
  Parents = Offspring;
end

% Step 5: Plot results
plot(1:NumGenerations, Propbw75)
xlabel('Generation number')
ylabel('Proportion of the bw75 allele')


% Step 6: Find out when (if ever) fixation took place
FixationGen = NumGenerations+1;
for i = NumGenerations:-1:1
    if Propbw75(i) == 0 | Propbw75(i) == 1
       FixationGen = i;
    end
end

if FixationGen == NumGenerations+1
   display('Fixation never took place!')
else
   display(['Fixation took place at generation',num2str(FixationGen)])
end
```

When you have written your code, see what happens to the simulated evolutionary outcomes when you change? (i) The population size (*N*); (ii) The initial frequency of the bw75 allele?

9. Suggested extensions

The following extensions are all quite difficult, and were designed for those with some programming experience. Ask for help if you get stuck!

1.
Modify your programme to simulate multiple replicates of Buri's experiments. Provide some summary statistics of the times to fixation, and the proportion of replicates where it is the bw75 allele that becomes fixed. Do these results match your expectations? (and if not, was your code wrong, or your expectations wrong?)

2.
Under the simple evolutionary model we have specified, the number of bw75 alleles in the next generation will be a binomially distributed random number, with "the number of trials" equal to *N* and "the probability of success" equal to *Propbw75*(generation-1). Modify your code to use this result directly. You should find that you don't need to keep track of the *N* population members. It should also speed things up for very large populations, but have no effect on the results.

Tip. The command `binornd` should be helpful.

3.
Buri's flies were diecious diploids. Modify your code to simulate these features of the real biology.

4.
Let us assume that having red alleles makes flies less attractive, such that they suffer a selection disadvantage of $s = 0.2$. How would you modify your code to incorporate this natural selection?