MODIFIED VERSION OF: An introduction to **Matlab** for dynamic modeling
***PART 3 ***

Stephen P. Ellner[1] and John Guckenheimer[2**]
[1]Department of Ecology and Evolutionary Biology, and
[2]Department of Mathematics
Cornell University

**** Modified by Eric Shea-Brown for AMATH 422/522:**
errors and other unpleasantries may have been introduced, unless you're in this class
refer to www.cam.cornell.edu/ $\sim$ dmb/DMBsupplements.html for the original!!
With additional material from Prof. Mark Goldman, UC Davis

1 While-loops

A while-loop lets an iteration stop or continue based on whether or not some condition holds, rather than continuing for a fixed number of iterations. For example, we can compute the solutions of a model until the time when some variable reaches a threshold value. The format of a while-loop is

```
while(condition);
    commands
end;
```

The loop repeats as long as the condition remains true. **Loop4.m**, on the website contains an example similar to the for-loop example; run it and you will get a graph of population sizes over time.

A few things to notice about the program:

1. First, even though the condition in the while statement said
   `while(popsize<1000)`
   the last population value was > 1000. That's because the condition is checked *before* the commands in the loop are executed. When the population size was 640 in generation 6, the condition was satisfied so the commands were executed again. After that the population size is 1280, so the loop is finished and the program moves on to statements following the loop.

2. Since we don't know in advance how many iterations are needed, we couldn't create in advance a vector to hold the results. Instead, a vector of results was constructed

==	Equal to
~=	Not equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
&	AND
|	OR
~	NOT

Table 1: Comparison and logical operators in Matlab.

by starting with the initial population size and appending each new value as it was calculated.

3. When the loop ends and we want to plot the results, the "y-values" are popsize, and the x values need to be x=0:something. To find "something", the **size** function is used to find the number of rows in popsize, and then construct an x-vector of the right size.

The conditions controlling a **while** loop are built up from operators that compare two variables (Table 1). Comparison operators produce a value 1 for true statements, and 0 for false. For example try

```
>> a=1; b=3; c=a<b, d=(a>b)
```

The parentheses around (a>b) are optional but improve readability.

More complicated conditions are built by using the **logical operators** AND, OR, and NOT to combine comparisons. The OR is **non-exclusive**, meaning that x|y is true if one or both of x and y are true. For example:

```
>> a=[1,2,3,4]; b=[1,1,5,5]; (a<b)&(a>3), (a<b)|(a>3)
```

When we compare two matrices of the same size, or compare a number with a matrix, comparisons are done element-by-element and the result is a matrix of the same size. For example

```
>> a=[1,2,3,4]; b=[1,1,5,5]; c=(a<=b), d=(b>3)
c =
     1     0     1     1
d =
     0     0     1     1
```

Within a while-loop it is often helpful to have a **counter** variable that keeps track of how many times the loop has been executed. In the following code, the counter variable is `n`:

```
n=1;
```

```
while(condition);
    commands
    n=n+1;
end;
```

The result is that `n=1` holds while the `commands` (whatever they are) are being executed for the first time. Afterward `n` is set to 2, which holds during the second time that the commands are executed, and so on. This is helpful, for example, if you want to store a series of results in a vector or matrix.

2 Random numbers

We'll start with the MATLAB command to make a single "pseudo" random number: `rand`.

- Type it and see what you get. Write it down.
- Quit matlab, then restart it. Repeat the above.
- Repeat this again ... this time, as soon as MATLAB begins, type `rand('state',sum(100*clock))`. That resets the "state" of the random number generator to a unique starting point that has to do with EXACTLY what time it is when you type it in. Thus, you'll end up with different random numbers each time ... as needed. CONCEPT: if you are trying to sample from the underlying probability distribution ALWAYS ALWAYS ALWAYS use this command before your first use of a random number generator.

Next question – how random ARE those random numbers?

Exercise 2.1

- Here is the MATLAB command to make a vector of n "pseudo" random numbers: `r_vector=rand(1,n).`
  Try it, for $n = 100$.
- Make a plot of these 100 random variables ... on horizontal axis, you should just have the integer 1 through 100. On the vertical, you should have a "*" above each of these numbers, giving the value of the corresponding random number.
- From our website, download and run `hist_demo.m` Modify this to take 10000 samples, and plot the probability density $p(x)$ using 100 histogram bins (remember, `help hist` is your friend!). Remember, you're going for the probability DENSITY, as for continuous-valued random numbers, and will need to properly normalize by the "bin width" used in plotting the histogram.
- Now, check your answer. You should have just computed the probability density $p(x)$ for a uniformly distributed random variable with range [0,1]. You will have just computed the values of that probability density at the centers c_j of all the histogram bins: specifically, you will have defined $p(c_j)$.

  How do you compute the probability that a continuous-valued random variable lies between a lefthand endpoint l and a righthand endpoint r? From class, recall that this is $\int_l^r p(x)dx$. How do we do this numerically, given the values $p(c_j)$? The answer comes from thinking back to calculus class, when we defined integrals as (Riemann) sums of bunches of rectangles, each with a height $p(c_j)$ and a width (Δ). Then, to compute an integral $\int_l^r p(x)dx$, we summed up the area of each rectangle that lay between our lefthand and righthand endpoints. That is, we computed $\sum_j p(c_j) \times \Delta$. This is the (simplest) way we compute integrals numerically as well.

  Use this method to compute the expected probability of the random variable lying between, say, 0.5 and 0.6? What should you get, and what does MATLAB return?

Exercise 2.2

- One of the concerns we had in discussing random number generators is that the each sequential random number is uncorrelated from the next – roughly speaking, that you can't predict the next random number from the previous ones! Please make a plot to check this by generating 1000 random numbers using MATLAB or R's `rand` command. Make the following scatter plot: where x_k is the k^{th} sample, plot x_k on the horizontal axis and x_{k+1} on the vertical, for $k = 1, ..., 1000$. Do you see any trends in your scatter plot? If you are curious, repeat this for the random number generator with $m = 31$ discussed in class. Do you see any correlations then?

2.1 Coin tossing

- Next, say we want to simulate the tossing of an unfair coin 1000 times, which comes up heads with probability, or frequency, p (a number between 0 and 1 that gives the

fraction of times that a heads occurs).

- Write a for loop with an if statement that turns `r_vector` into vector `heads_and_tails_vector` full of 0's and 1's, where a 1 corresponds to a coin toss that came out heads. Use $p = 0.5$. Then repeat with $p = 0.1$.
- Repeat with a single "vectorized" operation that does the same operation in a single line.

3 Simulating dwell times

We'll build up the tools we need for this, step by step.

Our objective is to generate synthetic data for single channel recordings from finite state Markov chains, and explore patterns in the data they produce.

The histogram of expected residence times for each **single** state in a Markov chain is exponential, with different mean residence times for different states. To observe this in the simplest case, we again consider coin tossing. Say you have a single coin that you're tossing, which has probability H of coming up "heads." You sit there tossing it over and over again. The two outcomes, heads or tails, are the different states in this case. Therefore the histogram of residence times for heads and tails should each be exponential. We will take the following steps to compute the residence times:

1. Generate sequences of independent coin tosses based on given probabilities.
2. Calculate the dwell times (also called residence times) by counting the number of tosses between each transition.

Let's get started with the first step. We'll call "heads" state 1, with value $S_1 = 1$, and likewise for tails and $S_2 = 2$.

- Write a code that "flips the coin 10000 times," producing a sequence of 10000 "1's" and "2's" that record the results, using $H = 0.6$. Call this list `states`. Please use a `for` loop, looping over the number of tosses, to accomplish this. If you are thinking that you could have accomplished this without a `for` loop, you are right – but we'll need this `for` loop structure for the more interesting cases that we will study below.

OK, now for the dwell times. Here is a strategy for writing the code to compute these. It is possible to come up with a more computationally efficient attack. Please code up the approach below, and then test any improvements you make to ensure that the results are the same!

- Next, we are going to make two lists of these transition times, one for each of the states.

Let us start out with these lists being empty, and we will build them up.

In MATLAB:

```
list_of_dwell_times_in_state_1=[];  %note: [] is the empty list
list_of_dwell_times_in_state_2=[];
```

Here's my strategy: I'm just going to loop down the states vector and keep track of what's happening. First, set `starting_state`: the state that I am dwelling in for the "current" sequence of states.

In MATLAB:

```
starting_state=states(1);
starting_timestep=1;  %the timestep at which I start dwelling.
```

Next, loop over all the states. We keep going until we come to a transition in the value of the state. When that happens, we store the number of steps that has elapsed since the last transition (or, for the first transition only, since we started counting) in one of the two lists made above. Here is the code, with comments in place:

In MATLAB:

```
for k=2:length(states)

    %Ask the question:  Am I still in the starting_state?
    if states(k) == starting_state
        %if so, do nothing
    else
        %OK, now I need to do something.  What's the dwell time in
        %starting_state?
        dwell_time=k-starting_timestep;

        %assign this dwell time to the right list.

        if starting_state==1
            list_of_dwell_times_in_state_1=[list_of_dwell_times_in_state_1 d
        else
            list_of_dwell_times_in_state_2=[list_of_dwell_times_in_state_2 d
        end

        %Finally, I must reset starting_state and starting_timestep to
        %begin measuring the NEXT dwell time

        starting_state=states(k);
        starting_timestep=k;

    end
```

```
end
```

- **Exercise 3.1** Cool! Now we're done. All that remains is to plot a histogram of dwell times in the various states. *Please assemble the code above, and add some lines to the end to compute the histogram of dwell times in state 1 and in state 2. Can you see that these dwell times reflect an exponential distribution? Please make the appropriate plot, taking a log to verify this. Can you estimate the value of H from the histograms alone, using a formula from class?*