The Hopfield Model of Associative Memory

In this laboratory, we will build a model of a recurrent network that can store and recall *associative memories*. Associative memory is a form of long-term memory (memory that lasts for anywhere from minutes to years) characterized by the property that presentation of only part of the memorized item can lead to recall of the entire item. Examples include the ability to recognize a person from seeing only a portion of their face, the ability to recall a memorized phone number when presented with just a few digits of the number, or the ability to recognize a place from a few environmental landmarks even though not every single detail of the place is the same as when you first visited it. In these examples, the key element underlying the ability to recall the memory is that one observes portions of the memorized item that are *associated with* the portions of the item not viewed. Such associations are hypothesized to be stored in the brain through recurrent connections between neurons. For example, in the hippocampal CA3 region, "place cells" that fire when one is at a particular spatial location in an environment are known to be connected together in a highly recurrent network, and there is some evidence that the recurrent connections between these neurons mediate the ability to recognize one's location in such an environment even when some of the landmarks in the environment have been removed.

I. The Hopfield Network architecture

The goal of the Hopfield network is to be able to correctly recall one of NumPatt "memory" patterns $\mathbf{Mem}^{(a)}$, a=1,2,...NumPatt, when presented with a stimulus pattern that is close to one of the memory patterns. The memory patterns each consist of N elements whose values are either +1 or -1. For example, if the memory patterns were faces drawn in black-and-white, then N might represent the number of pixels in the drawing of the face, a pixel value of +1 might represent a white pixel, and a pixel value of -1 might represent a black pixel.

The memory patterns are to be recalled by a network of N "neurons" that are interconnected with connections of strength W_{ij}=the strength of the synapse from "neuron" j to "neuron" i. The reason "neuron" is inside quotation marks is that the Hopfield model assumes an extremely simplified 2-state model of the neuron in which each neuron is deemed to be either in an active state or an inactive state. We will denote the activity of the i^{th} neuron in the network by x_i, and let $x_i = +1$ denote an active neuron and $x_i = -1$ denote an inactive neuron.

To test the network's recall ability, we will present the network with an initial pattern that drives the network into an initial activity state $\mathbf{x}(t=0)$. The network's recall will be deemed successful if, for all times t greater than some value, the network state $\mathbf{x}$ equals (or is very close to) the memory pattern that is closest to the initial activity state. For example, if the closest memory pattern to the initial state were $\mathbf{Mem}^{(1)}$, then we would say that recall is successful if $\mathbf{x}(t)= \mathbf{Mem}^{(1)}$ for all times t > some value.

To summarize the notation from this section:

NumPatt = number of memory patterns
N = number of neurons and also number of elements in a memory pattern
$\mathbf{Mem}^{(a)}$, a=1,2,...NumPatt, = the a^{th} memory pattern vector of +1's and -1's
Mem_i^a = the i^{th} element of the a^{th} memory pattern (i=1,2,…N)
$\mathbf{x}(t)$ = activity pattern of the neurons in the network at time t (+1=active,-1=inactive)
$x_i(t)$ = activity of the i^{th} neuron in the network at time t (i=1,2,…N)
W_{ij} = the strength of the synapse from "neuron" j to "neuron" i

II. Network dynamics

The dynamics of the network occurs in discrete time steps with the elements x_i of **x**(t) being updated according to the rule:

$$x_i(t) = \mathrm{sgn}\left(\sum_{j=1}^{N} W_{ij} x_j(t)\right) \tag{1}$$

where the function sgn is defined as

$$\mathrm{sgn}(z) = \begin{cases} +1 \text{ if } z \geq 0 \\ -1 \text{ if } z < 0 \end{cases}$$

Here one can think of $\sum_{j=1}^{N} W_{ij} x_j(t)$ as the total "current" being input to neuron x_i from its neighbors.

III. Value of the weight matrix W_{ij}: The Hebb Rule

The key to making the network capable of retrieving a memory pattern is to have correct network weights W_{ij}. These weights are constructed from an application of the "Hebb rule" that says that "Neurons that fire together wire together (i.e. are connected by a positive strength connection") and neurons that do not fire together form an inhibitory connection. We can imagine that the network was trained by being presented with the various patterns $\mathbf{Mem}^{(a)}$ over and over again (with each presentation causing the activity of the network to equal the memory pattern, $\mathbf{x}=\mathbf{Mem}^{(a)}$). At the end of such training, the weights W_{ij} will equal a sum of terms $Mem_i^a\ Mem_j^a$ corresponding to whether neurons i and j had the same activities (either both +1 or both -1, so that the product = +1) or opposite activities (so that the product = -1). Summing over all trained memory patterns then gives

$$W_{ij} = \sum_{a=1}^{NumPatt} Mem_i^a Mem_j^a \text{ if } i \neq j \tag{2}$$

We assume that neurons do not connect to themselves:

$$W_{ii} = 0$$

IV. Quantifying network performance

To quantify how well the network is performing in retrieving a memory pattern, say $\mathbf{Mem}^{(1)}$, we need a measure of how close the activity state **x**(t) is to the memory pattern at any time. This can be measured with the overlap (or 'correlation') function

$$q(t) = \frac{1}{N}\sum_{i=1}^{N} x_i Mem_i^1 \tag{3}$$

This function will equal 1 if $x_i = Mem_i^1$ for all elements i and will equal zero if there is no similarity between the state of the network **x** and the memory vector $\mathbf{Mem}^{(1)}$.

V. Constructing a model that retrieves 1 memory

In this tutorial, we will build a model of N=100 neurons and see how many memory patterns it can successfully retrieve. The maximum number of memory patterns that a network can retrieve is known as the network's *capacity*. We first will construct a model that (hopefully) retrieves a single memory pattern. Then, we will increase the number of memory patterns until we find that the network is no longer capable of accurately retrieving all of the memories.

Let's get started by opening a file, adding some comments, clearing all variables, and closing all figure windows:

```
%BUILD A HOPFIELD NETWORK WHICH RETRIEVES RANDOM MEMORIES

clear all;  %clear all variables
close all; %close all open figure windows
```

Let's next define some simulation parameters such as the number of neurons in the network, the number of memory patterns, and the number of time steps for which we would like to run our network. For now, let's show that the network can successfully retrieve 1 pattern by setting the number of patterns NumPatt=1

```
%DEFINE NETWORK PARAMETERS
NumPatt = 1  %Number of memory patterns that network will attempt to store and retrieve
N = 100  %Number of neurons in the network (and number of elements in a memory pattern)
NumTimeSteps = 20 %length of run
```

Next we need to define the memory patterns and, from the Hebb rule, the corresponding network weight matrix W_{ij}. To assign the memory patterns randomly, we take advantage of the **rand(N,M)** command which generates an NxM array whose elements are random numbers between 0 and 1:

```
%MEMORY PATTERNS
Mem_mat = 2*round(rand(N,NumPatt))-1; %random strings of 1's and -1's
```

Note that the **round** command rounds off the random number, giving a 1 with probability 0.5 (i.e. if 0.5 <= rand < 1) or a 0 with probability 0.5 (if 0 < rand < 0.5). Multiplying 0 or 1 by 2 and then subtracting 1 gives either -1 or +1, respectively. Thus, this line defines a matrix whose NumPatt columns each contain an N-element vector of random memory patterns.

We next will use the Hebb rule to define the synaptic weights. Re-writing equation (2) into the row and column notation for our memories, we have

$$W(i,j) = \sum_{a=1}^{NumPatt} Mem(i,a)Mem(j,a) = \sum_{a=1}^{NumPatt} Mem(i,a)Mem^T(a,j)$$

where the superscripted *T* denotes the transpose of the matrix (i.e. the matrix obtained by making making the nth row of the first matrix into the nth column of the transposed matrix; e.g. the transpose of [1 2 3;4 5 6;7 8 9] would be [1 4 7;2 5 8;3 6 9]).

The above equation then describes matrix multiplication of Mem_mat by its transpose:

```
%DEFINE SYNAPTIC WEIGHT MATRIX
W_mat = Mem_mat*Mem_mat';
```

Now that we have memory patterns and a network with connections defined by the Hebb rule, we need to define the initial pattern of activity of the network before seeing whether this pattern converges to one of the stored memory patterns.

Let's construct an initial activity pattern **x**(t=0) that approximately overlaps the first memory pattern $\mathbf{Mem}^{(1)}$ = Mem_mat(:,1). In particular, let's construct an initial activity pattern **x**(t=0) that has random elements chosen so that (on average if we did this many times) the overlap with $\mathbf{Mem}^{(1)}$ will attain a value qStart.

This can be achieved by using the following procedure: For each element i, set $x_i(0)$ = Mem(i,1) with probability qStart (this makes a fraction qStart of the elements align with $\mathbf{Mem}^{(1)}$ on average and can be accomplished by asking whether a number generated randomly from rand is less than qStart). With probability 1 – qStart, set $x_i(0)$ randomly (with 50% probability) to either +1 or -1 (i.e. the remaining elements will, on average, have 0 overlap with $\mathbf{Mem}^{(1)}$). The following code accomplishes this task for a value qStart=0.7:

```
%INITIALIZE NETWORK ACTIVITY
qStart = 0.7; %average overlap of initial activity pattern with first memory (the one to be retrieved)
for i=1:N
   if rand < qStart %assign x_i equal to Mem(i,1)
      x_vect(i)=Mem_mat(i,1);
   else %assign random value for x_i
      if rand < .5
         x_vect(i) = 1;
      else
         x_vect(i) = -1;
      end
   end
end
```

We next want to run the network dynamics (determined by the network update rule) and compute the overlap of x(t) with $\text{Mem}^{(1)}$. First, let's compute this overlap for x(t=0) using the x_vect values initialized by the code preceding this paragraph:

```
%compute initial overlap
q_vect = zeros(1,NumTimeSteps);
q_vect(1) = x_vect*Mem_mat(:,1)/N;
```

The first line initializes the overlap vector q_vect, where the elements of q_vect denote the overlap at a given time step. The second line computes the overlap according to equation (3).

Good! Now we are ready to run the model by programming its dynamics.

VI. Running the model that retrieves 1 memory

To run the model, we need a for loop over the time steps. Inside this for loop should be an application of the update rule (equation (1)) followed by a computation of the overlap q at this time step. This can be accomplished as follows:

```
%RUN MODEL AND COMPUTE OVERLAP AT EACH TIME STEP
for tStep=2:NumTimeSteps
   x_vect = (2*(W_mat*x_vect' >= 0) - 1)'; %update rule
   q_vect(tStep) = x_vect*Mem_mat(:,1)/N; %compute overlap of newly updated x_vect with Mem1
end
```

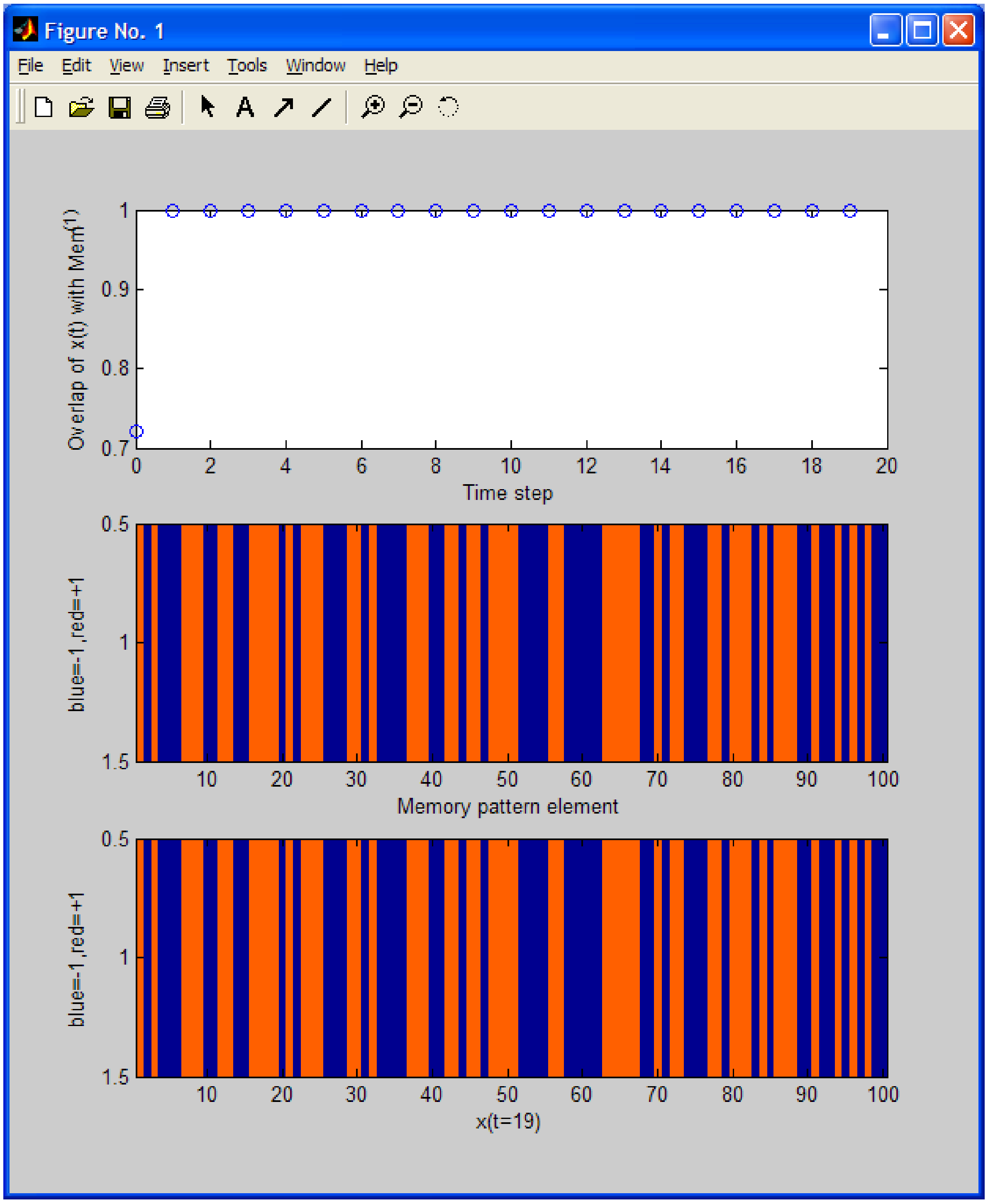
Note the logic of the first line within the for loop. The term in parentheses (W_mat*x_vect' >= 0) returns a value of 1 (=TRUE) if the input current $I = \sum_{j=1}^{N} W_{ij} x_j(t)$ = W_mat*x_vect' is greater than or equal to zero and returns a value of 0 (=FALSE) if the input current is less than zero. Multiplying the 1 or 0 returned by this expression by 2 and subtracting 1 then gives a value of +1 or -1, respectively.

Finally, we are ready to plot our results. Let's make 3 subplots: the first will show the overlap q(t). The second will show the pattern Mem_mat(:,1) that the network is attempting to retrieve. The third will show the value of **x** at the end of the run, which we can then visibly compare with Mem_mat(:,1) to get a more intuitive sense of how much **x** overlaps the desired memory pattern. This is done as follows:

```
figure(1)
subplot(3,1,1)
plot(0:NumTimeSteps-1,q_vect,'.')
xlabel('Time step')
ylabel('Overlap of x(t) with Mem^{(1)}')
subplot(3,1,2)
image(50*Mem_mat(:,1)')
xlabel('Memory pattern element')
ylabel('blue=-1,red=+1')
subplot(3,1,3)
image(50*x_vect)
xlabel(strcat('x(t=',num2str(NumTimeSteps-1),')'))
ylabel('blue=-1,red=+1')
```

Here, the **image**(*vector*) command generates a sequence of vertical bars whose colors are determined by the value of the corresponding element of the *vector* argument to this command. Note also that ^**{**(1)**}** in the text of the ylabel command creates a superscripting of everything enclosed by the curly braces. Finally, notice in the 2nd to final line that we have used the command **num2str** to convert the number that equals NumTimeSteps-1 (=19 in our case) to a text "string" (I don't know why computer scientists call sequences of text characters "strings" but they do!). The command **strcat**, which is short for "concatenate (i.e. combine together) strings" then assembles the 3 text strings given into a single long text string that is plotted as the x-axis label.

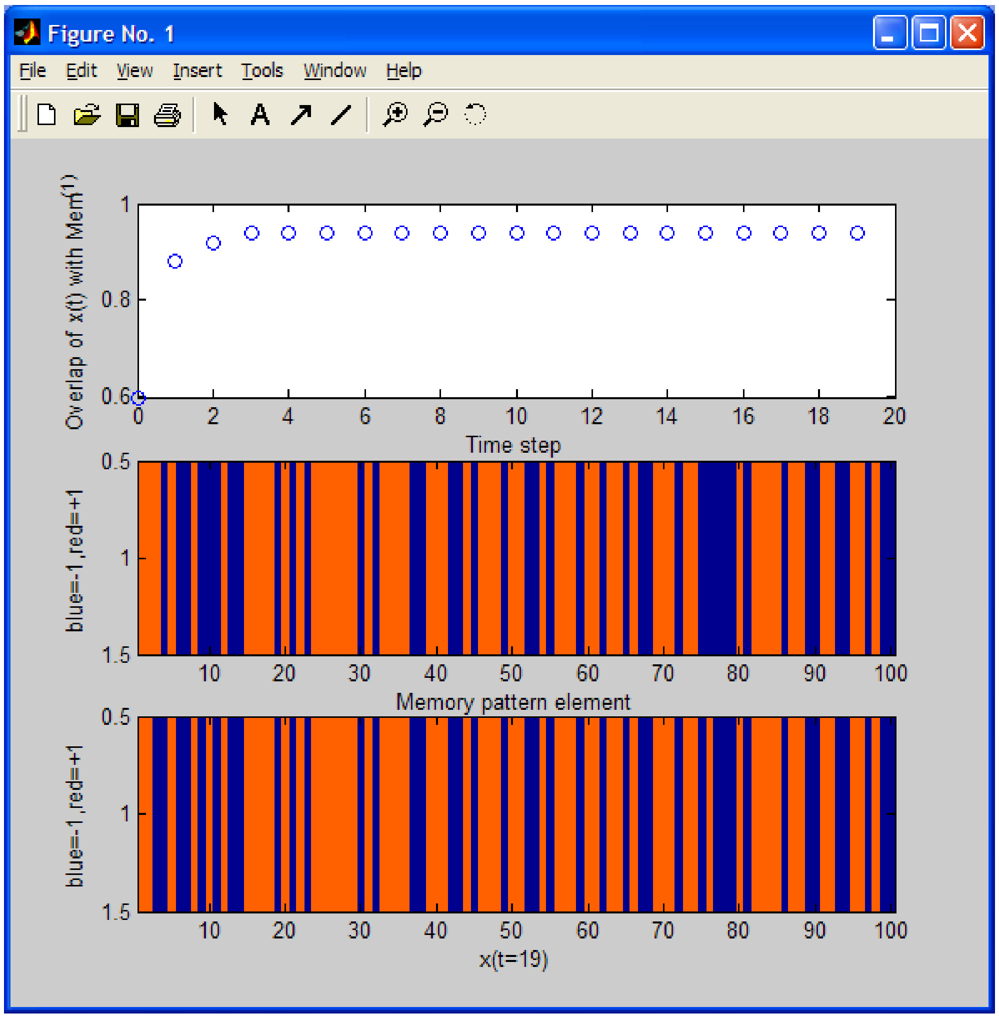
Now run your code and you should get something like the following result (the top panel should be nearly identical, except for the initial point, to that shown below. The following two panels may differ in detail because they reflect the values your rand command randomly generated):



Notice from the top panel that the network activity pattern **x** converged to exactly overlapping the memory pattern **Mem**$^{(1)}$ within a single time step! You can directly compare the memory pattern and network activity pattern at the end of the run (t=19) in the 2nd and 3rd panels, respectively.

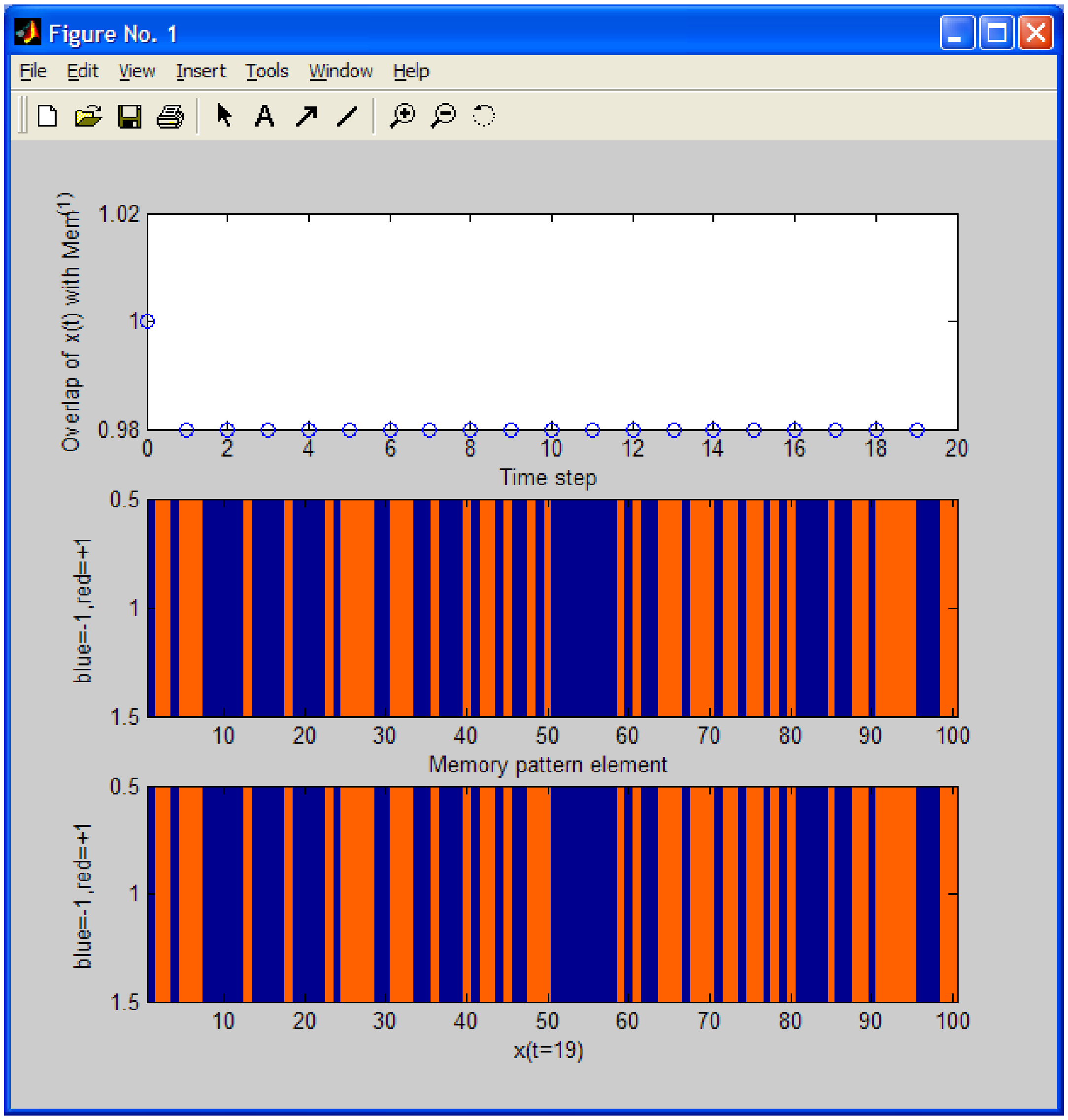
VII. Running the model with many memory patterns to retrieve

You have now finished all of the programming. The final thing to do is to "play" with the model to see how many memories it can successfully store. To do this, try raising the number of patterns to a larger number (say NumPatt=20) and run it several times (maybe 10 times). There will be quite a bit of variability from run to run but a typical run should look something like the following:



We see that now the network only partly succeeds in recovering the memory pattern. You can directly see which elements it "missed" by comparing the bottom two subplots.

Another fundamental question to address is whether the model will stay at a memory pattern if it starts at such a pattern—this is not guaranteed! Let's test this for NumPatt = 20 by changing qStart to 1 and running the model several (10 or 20) times. You will likely find that it often does succeed in staying at the memorized pattern, but *not always*! A typical erroneous run will likely not having many elements mixed up, such as the following run with overlap of 0.98 (=1 element flipped; remember that an overlap of 0 would be half, or 50, of the elements being flipped):



If you are searching for it, the flipped element is element # 49.

As an exercise, you can play with the network for different values of NumPatt and see how many memory patterns the network can successfully retrieve if, for example:

a) it starts at the memory pattern **Mem**$^{\mathbf{(1)}}$ (i.e. qStart=1)

or

b) it starts with qStart = 0.7 (approximately 15 elements flipped)

These are both measures of the storage *capacity* of the network.