MATLAB Refresher

Problems by Emily Mackevicius
Adapted from Woods Hole and MIT
Computational Neuroscience courses

Instructions: Choose several parts to work on. TAs will be there to help!

Part 0: Install MATLAB, get familiar with basics, how to look stuff up

Download and install MATLAB (you'll need a license and it can take a while to download and install so make sure you have MATLAB installed ahead of time):

- http://www.mathworks.com/products/matlab/whatsnew.html

Good basic tutorial by Mark Goldman:

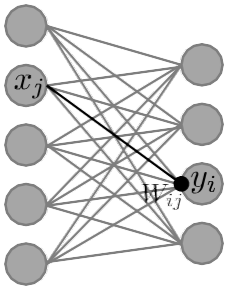
https://goldmanlab.faculty.ucdavis.edu/teaching/

MATLAB documentation

- There's lots of help online. For documentation on a particular function (eg `rand`) type `help rand` or `doc rand` into MATLAB's command window.

Part 1: Matrix operations for a feedforward network

We will construct a 2 layer feedforward linear network, and use matrix operations to calculate its outputs, given its inputs and weights. We'll call the output neurons $y_1, \ldots, y_M$ and input neurons $x_1, \ldots, x_N$. W_{ij} is the connection strength (weight) onto neuron y_i from neuron x_j. We refer to W as the weight matrix.



1) What does each row of the weight matrix represent? Each column? row: the response of the receiving node y_i
column: the contribution of the sending node x_j
2) Write an expression for calculating y_i, the response of the ith output neuron. x*w(j-th row)
3) Write an expression for calculating the contribution of input neuron x_j to the network output (this should be a vector of length M). Note that the total network output is the sum of the contributions from each input neuron. w(i-th column) * y

Matrix notation gives us a compact way of expressing all of this information:

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{pmatrix} = \begin{pmatrix} W_{11} & W_{12} & \cdots & W_{1N} \\ W_{21} & W_{22} & \cdots & W_{2N} \\ \vdots & \vdots & & \vdots \\ W_{M1} & W_{M2} & \cdots & W_{MN} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix}$$

4) Convince yourself that the equation above captures what you found in parts 2) and 3).

Now we'll use MATLAB to construct an example network, with $N = 50$ and $M = 10$.

5) Generate a weight matrix. Assume the weights are random and uniform between -1 and 1 (use `rand`).
6) Generate a 50-dimensional pattern of inputs consisting of Gaussian entries (use `randn`).
7) Calculate the network output.

Part 2: Logical operations, for-loops, and plotting (random walk)

We will construct a biased random walk with a reflecting barrier and reset. You can think of this as a simple model of the voltage of a neuron. The voltage increases, with some fluctuations, until it reaches a threshold, at which point it spikes then resets to a resting value, and starts the process again.

Specifically, the voltage of the cell is given by: $V(t+1) = V(t) + dV(t)$, where $dV(t)$ is 1 with probability p, and -1 with probability $1 - p$. We want the system to reset once it hits a maximum ceiling, so we will add the condition: $V(t+1) = Vreset$ if $V(t) > Vthres$.

1) Make a function `GenerateVoltage(p,T,Vreset,Vthresh,V0)` that takes as input the probability of going up, `p`, the number of time steps to simulate, `T`, the reset voltage, `Vreset`, the spike threshold, `Vthres`, and the initial voltage, `V0`. Your function should return a vector V, the voltage at all time steps from 1 to T. To create the function, make a new file *GenerateVoltage.m* with the following syntax:

```
function V = GenerateVoltage(p,T,Vreset,Vthresh,V0)
     % your code here
end
```

2) Inside the function, create the vector V = zeros(1,T); that will hold voltage values at each step of the process:

```
for t = 1:T-1
     % update V(t+1)
end
```

3) Run your function to simulate a neuron with initial voltage of -65mV, threshold of -45mV, and reset voltage of -70mV. Choose p so that your neuron has an average firing rate of approximately 10Hz, assuming each time step corresponds to 1ms (feel free to try several values of p). Calculate the voltage values for 1 second
4) Plot the voltage as a function of time using `plot`. Label your axes using `xlabel` and `ylabel`.

Part 3: Convolution to estimate voltage response to a spike train

A convolution is a mathematical operation on two functions that expresses the amount of overlap of one function as it is shifted over the other. Mathematically, it is defined as

$$[f * g](t) = \int_{-\infty}^{+\infty} f(\tau)g(t-\tau)d\tau$$
$$= \int_{-\infty}^{+\infty} g(\tau)f(t-\tau)d\tau$$

We will use convolution to estimate the voltage response of a neuron to an incoming spike train.

1) Generate 3 seconds of a Poisson spike train with firing rate 20Hz. Use `spiketrain = rand(1,N)>(1-p);` where N is the total number of time steps (each time step should 1ms) and p is the probability of spiking in any given time step (you need to calculate N and p).
2) Construct a kernel, the response of the neuron to one spike at t=0. We assume the neuron is linear, that is, the response to multiple spikes is the sum of the responses to each individual spike. For the kernel, use an exponential with mean mu of 5ms. Calculate the kernel for values between -50 and 50ms: `k = exppdf(-50:50, mu);` Plot the kernel.
3) Now estimate the voltage response of the cell to the spike train by convolving your spike train from 1) with the kernel from 2). Use `conv` (but first, convert `spiketrain` to a double from a logical: `spiketrain = double(spiketrain);`.
4) Plot the voltage and the spike train on two separate panels, using `subplot`. Make sure to align them properly in time (type `doc conv` to see how `conv` works). Zoom in to see what happens to the voltage when incoming spikes occur in rapid succession. It may help to use `linkaxes` to align the two panels when you zoom.

Part 4: Convolution to detect edges in images

In an image, edges are where the image is different from its neighbors. Convolution in two dimensions is often used for edge detection in image processing. The output of this operation is very similar to the response of cells in primary visual cortex, which respond selectively to oriented edges. The following kernel will be zero in regions of the image where neighboring pixels have the same value, and nonzero for edges:

```
k = [0 0 0; 0 1.125 0; 0 0 0] - .125*ones(3,3);
```

- Load the `octopus` matrix file from the Tutorial section on MCN 2025 PBworks. Plot it using `imagesc(octopus); colormap gray;`

1) Use `conv2` to convolve the image with the kernel `k`, and plot the result using `imagesc`. Notice that edges can be darker and/or lighter than the gray background.

2) Plot the absolute value of the convolution (this will show both positive and negative edges as lighter than the background). You've built a simple edge detector!