We will be using Matlab for the programming assignments in this class. Here we provide a basic introduction to Matlab that will be useful for completing your programming assignments. Matlab is a scripting language and is very commonly used tool for numerical computation, visualization, and programming.

**General Matlab**

- Basic variable type is a matrix:

  ```matlab
  >> M = [11, 12, 13; 14, 15, 16]
  M =
    11  12  13
    14  15  16
  ```

  The column entries are separated by spaces or commas (,) while ; is used to separate rows in the matrix definition. If there is only one element in the matrix, it is a scalar variable. If there is only one row or one column, the matrix is called a vector. Signals are often defined as vectors or arrays. Another way to define a vector is by using the colon (:) operator:

  ```matlab
  >> a = 1:2:10
  a =
    1  3  5  7  9
  ```

  Here the vector is generated starting at 1 with the difference between each element as 2 up until 10 is reached. The middle value, called skip parameter, is optional and default value is 1.

- Size and length:
As seen above, size returns the row size followed by the column size of M. Length returns the number of elements in a vector.

• Indexing:

In Matlab, indexing starts at 1 unlike many other programming languages. For the matrix M defined above, M(2,3) is the element at the 2nd row and 3rd column of M and will refer to the value 16. To index a range of values, the colon operator can be used. For example:

\[
\text{>> } \text{M}(1, 2:3) \\
\text{ans =} \\
\begin{bmatrix}
12 & 13 \\
\end{bmatrix}
\]

\[
\text{>> } \text{M}(1, :) \\
\text{ans =} \\
\begin{bmatrix}
11 & 12 & 13 \\
\end{bmatrix}
\]

Again, like the colon operator used for generating vectors, the indexing can also skip elements at equal steps or even reverse index the elements.

\[
\text{>> } \text{M}(1, 3:-1:2) \\
\text{ans =} \\
\begin{bmatrix}
13 & 12 \\
\end{bmatrix}
\]

• Find: find(condition) returns the indices of elements in the matrix that satisfy the given condition. For example:
Here, we find the elements in a vector that are positive without explicitly stating the indices.

- **Reshape**: `reshape(A, row_size, column_size)` – this function can reshape the matrix to a new shape as long as the number of elements are same in the old matrix and new matrix.

- **help command**: Provides the documentation for the associated key word.

```matlab
>> a = -5:1:5
a =
    -5    -4    -3    -2    -1     0     1     2     3     4     5

>> find(a>0)
ans =
     7     8     9    10    11

>> positives = a(find(a>0))
positives =
     1     2     3     4     5
```

As you can see above, there are many ways find function can be used. When unsure about a command, use `help`.
• Matrix multiplication: Let $A$ be a $m \times n$ matrix and $B$ be a $n \times p$ matrix. Then the matrix multiplication $C = AB$ can be performed in Matlab by using the * operator:

$$
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
\end{bmatrix} \ast \begin{bmatrix}
5 \\
1 \\
2 \\
\end{bmatrix} = \begin{bmatrix}
13 \\
37 \\
\end{bmatrix}
$$

Thus, here $A$ is a $2 \times 3$ matrix and $B$ is a $3 \times 1$ vector. Therefore, the result of matrix multiplication stored in $C$ is a $2 \times 1$ vector.

• Point-wise Array Operations: Multiplication is performed element by element, so they must have same size in both dimensions.

$$
\begin{bmatrix}
1 & 2 \\
4 & 5 \\
\end{bmatrix} \ast \begin{bmatrix}
1 & 2 \\
4 & 5 \\
\end{bmatrix} = \begin{bmatrix}
1 & 4 \\
16 & 25 \\
\end{bmatrix}
$$

As you might notice, we cannot perform matrix multiplication for this case as the number of columns in the first matrix is not the same as number of rows in the second matrix. Here, point-wise multiplication is performed where
each element in the first matrix is multiplied with the corresponding element in the second matrix (in this case, itself). Although matrix multiplication is not commutative, point-wise multiplication is commutative.

Any time a point (.) is used along with an arithmetic operation, Matlab performs point-wise operations. For example:

\[ xx = (0.9)^{0:49} \]

will generate a vector with 0.9 raised to the power of 0,1,...,49. Hence, point-wise operation (also called vectorization) makes it possible to write code without loops and they are much more efficient. Therefore, we must make all the effort to avoid loops as much as possible.

- Plotting: `Plot(x,y)` – x is a vector that defines the x-axis values and y is a vector that defines the y-axis values. Plot function plots these set of points and connects them together using linear interpolation.
- Stem(x,y) – plots discrete time stem plots with the heights associated stored in the y vector and the x vector stores the x-axis positions.

- Scripts - Commands stored in a m-file.

- Writing a Matlab function: Let’s write function that that takes a vector and find its mean and standard deviation:

```matlab
function [mean, stddev] = stat(x)

%STAT Interesting statistics.

n = length(x);
mean = sum(x) / n;
stddev = sqrt(sum((x - mean).^2)/n);

end
```

The first line of this code shows how to define a function using the function keyword. All the variables defined within the function are only available in workspace while the function is running. Matlab does not require a return statement. As long as the output variables are defined, the function will return the final values stored in the output variables.

- For loop:

The definition uses the colon operator to define the range of values to loop over. The example below shows how to write a simple for-loop to find sum of elements in a vector.

```matlab
sum = 0;
for i = 1:length(x)
    sum = sum + x(i);
end
```
Signal Processing related

Creating signals as arrays

By creating a time array, and then passing that vector to a function such as `sin()`, we can apply that function to each element in our time array. For example:

```matlab
time = 0:.001:1;
sig = sin(2*pi*time);
plot(time,sig)
```

In this case, we made time be from 0 to 1, in increments of 0.001. We then made a sine wave on this interval, and plotted it. The plot shows

Note that while this looks like a continuous sine wave, since it is in a computer it is discrete. However, since the rate is so high, it approximates continuous time well. Lets make another sine wave, over a longer interval, that we can play on our computer speakers.

```matlab
time = (0:8192) / 8192;
tone1 = sin(2*pi*400*time);
sound(tone1,8192);
```

We constructed a new time vector, that represents time in seconds, sampled at 8192Hz. This is chosen as that is a standard way to represent sound on a computer. `tone1` is a sin wave at 400Hz, sampled at 8192Hz. the `sound` function will play the signal over the computer speakers.

Fourier Transforms

Using the last example, we can now take a look at the spectrum of a sin wave.

```matlab
spec = fft(tone1);
spec_centered = fftshift(spec);
freqs = -8192/2:8192/2;
plot(freqs,abs(spec_centered));
```
**fft** computes the Discrete Fourier Transform on our signal **tone**. However, since our sampling rate is high, this approximates a Continuous Time Fourier Transform. It returns an array that is the same length as the input.

**fftshift** is an additional function that is used to shift the array returned by **fft** such that the negative frequencies are on the left, the positive frequencies are on the right, and 0 is in the middle.

**freqs** is simply an array that gives the frequencies in Hz corresponding to the elements of **spec_centered**.

Lastly, we call **abs** on **spec_centered** so that we plot the magnitude. The plot we obtain is:

![Plot](image)

Here we see two “impulses” at -400Hz and 400Hz, as we should for a sine wave at 400Hz. Note that they do not actually go to infinity, but this is approximate as everything is actually being done in discrete time.

Let’s do another example, with sinc, instead.

```matlab
    time = (-4096:4096) / 8192;
sig = sinc(2*1000*time);
spec = fft(sig);
spec_centered = fftshift(spec);
freqs = -8192/2:8192/2;
plot(freqs,abs(spec_centered));
```

Note that this time we start **time** at a negative value, so that time 0 is in the center. Also, we chose a frequency of 1000Hz to be the cutoff. There is no π this time, due to MATLAB’s definition of sinc.
The resulting plot is:

Window Functions

In this course we often use window functions to reduce things like gibbs phenomena in our signals. The \texttt{hann(N)} function returns an \(N\)-length array of the hann window, shown below.

\begin{verbatim}
   h = hann(8193)';
   plot(h)
\end{verbatim}

Note that we used the \texttt{'} operator to take the transpose of the array, as \texttt{hann(N)} returns an \(N \times 1\) array, and we want \(1 \times N\).
Here is a simple example building off of the previous sinc example, but with a hanning window used.

```matlab
time = (-4096:4096) / 8192;
sig = sinc(2*1000*time);
spec = fft(sig .* h);
spec_centered = fftshift(spec);
freqs = -8192/2:8192/2;
plot(freqs,abs(spec_centered));
```