Vectors and Plotting

MATH-151: Mathematical Algorithms in Matlab

September 6, 2023

What is a Vector?

- For coding, a vector is an ordered collection of values.

  Ex: `v = [1 0 -1 0 1 0 -1 0 1]`

- This will help us keep track of a whole lot of things all at once without having to make up new variable names for each value!

What is a Vector?

- For coding, a vector is an ordered collection of values.

  Ex: `v = [1 0 -1 0 1 0 -1 0 1]`

- This will help us keep track of a whole lot of things all at once without having to make up new variable names for each value!
- There are a few different ways we can think about or use vectors in code
  - Data sets: Vectors can be used to store data sets for a common variable, with the order representing which subject that sample was taken

    `height = [68 62 73 67 76]`

What is a Vector?

- For coding, a vector is an ordered collection of values.

  Ex: `v = [1 0 -1 0 1 0 -1 0 1]`

- This will help us keep track of a whole lot of things all at once without having to make up new variable names for each value!
- There are a few different ways we can think about or use vectors in code
  - Data sets: Vectors can be used to store data sets for a common variable, with the order representing which subject that sample was taken

    `height = [68 62 73 67 76]`
  - Discretization of a function: Since a continuous function is made up of infinite values, we can use vectors to represent our functions at a number of known points

    if `x = [0 pi/2 pi 3*pi/2 2*pi]`, then `cos(x) = [1 0 -1 0 1]`

What is a Vector?

- For coding, a vector is an ordered collection of values.

  Ex: `v = [1 0 -1 0 1 0 -1 0 1]`

- This will help us keep track of a whole lot of things all at once without having to make up new variable names for each value!
- There are a few different ways we can think about or use vectors in code
  - Data sets: Vectors can be used to store data sets for a common variable, with the order representing which subject that sample was taken

    `height = [68 62 73 67 76]`
  - Discretization of a function: Since a continuous function is made up of infinite values, we can use vectors to represent our functions at a number of known points

    if `x = [0 pi/2 pi 3*pi/2 2*pi]`, then `cos(x) = [1 0 -1 0 1]`
  - Physical/Mathematical sense: We won't use this as much, but we can also represent vectors as a physical magnitude and direction, or the output/solution to a linear system

Generating Special Vectors

- Zeros and Ones: We can use the `zeros` or `ones` functions to initialize a vector or zeros or ones.

```
>> zeros(1,5)
ans =
     0     0     0     0     0
>> ones(1,7)
ans =
     1     1     1     1     1     1     1
>> 24*ones(1,6)
ans =
    24    24    24    24    24    24
```

Generating Special Vectors

- Zeros and Ones: We can use the `zeros` or `ones` functions to initialize a vector or zeros or ones.

```
>> zeros(1,5)
ans =
     0     0     0     0     0
>> ones(1,7)
ans =
     1     1     1     1     1     1     1
>> 24*ones(1,6)
ans =
    24    24    24    24    24    24
```

- Uniformly Spaced Vectors: If we have end points and a step size, we can create a vector using `start_pt:step_size:end_pt`. Or if we want a number of points we can use `linspace`

```
>> x = 2:0.5:5
x =
    2.0000    2.5000    3.0000    3.5000    4.0000    4.5000    5.0000
>> x = linspace(2,5,6)
x =
    2.0000    2.6000    3.2000    3.8000    4.4000    5.0000
```

Generating Special Vectors

- Zeros and Ones: We can use the `zeros` or `ones` functions to initialize a vector or zeros or ones.

```
>> zeros(1,5)
ans =
     0     0     0     0     0
>> ones(1,7)
ans =
     1     1     1     1     1     1     1
>> 24*ones(1,6)
ans =
    24    24    24    24    24    24
```

- Uniformly Spaced Vectors: If we have end points and a step size, we can create a vector using `start_pt:step_size:end_pt`. Or if we want a number of points we can use `linspace`

```
>> x = 2:0.5:5
x =
    2.0000    2.5000    3.0000    3.5000    4.0000    4.5000    5.0000
>> x = linspace(2,5,6)
x =
    2.0000    2.6000    3.2000    3.8000    4.4000    5.0000
```

- Random Vectors: Sometimes we want to apply random noise to a vector (happens a lot in simulation), we can generate a uniform random variable using `rand`, or normally-distributed using `randn`

```
>> rand(1,5)
ans =
    0.4945    0.0373    0.2883    0.8890    0.5616
>> 5*randn(1,5) + 68
ans =
   71.5870   70.7059   63.5368   72.4473   71.7677
```

Using Vectors

- Vectors are composed of **elements**, each of which has a corresponding **index** from 1 to N. We can extract a specific value by knowing its index.

```
>> x = [5 4 8 1];
>> x(1)
ans =
     5
>> x(3)
ans =
     8
```

Using Vectors

- Vectors are composed of **elements**, each of which has a corresponding **index** from 1 to N. We can extract a specific value by knowing its index.
- We can do arithmetic with either a **scalar** or element-wise with a vector of the same size.
  - (Note: For multiplying two vectors we use `.*` instead of just `*`, the same for `.^` when doing powers)

```
>> x = [5 4 8 1];
>> x(1)
ans =
     5
>> x(3)
ans =
     8
```

```
>> x + 2
ans =
     7     6    10     3
>> 3*x
ans =
    15    12    24     3
>> y = [1 2 3 4];
>> x + y
ans =
     6     6    11     5
>> x.*y
ans =
     5     8    24     4
```

Using Vectors

- Vectors are composed of **elements**, each of which has a corresponding **index** from 1 to N. We can extract a specific value by knowing its index.
- We can do arithmetic with either a **scalar** or element-wise with a vector of the same size.
  - (Note: For multiplying two vectors we use `.*` instead of just `*`, the same for `.^` when doing powers)
- Many mathematical functions we know will also operate on element-wise.

```
>> x = [5 4 8 1];
>> x(1)
ans =
     5
>> x(3)
ans =
     8
```

```
>> x + 2
ans =
     7     6    10     3
>> 3*x
ans =
    15    12    24     3
>> y = [1 2 3 4];
>> x + y
ans =
     6     6    11     5
>> x.*y
ans =
     5     8    24     4
```

```
>> x = [0 1 2];
>> exp(x)
ans =
    1.0000    2.7183    7.3891
>> log(x)
ans =
      -Inf         0    0.6931
>> sin(pi*x)
ans =
   1.0e-15 *
         0    0.1225   -0.2449
>> x.^2
ans =
     0     1     4
```

Using Vectors

- Vectors are composed of **elements**, each of which has a corresponding **index** from 1 to N. We can extract a specific value by knowing its index.
- We can do arithmetic with either a **scalar** or element-wise with a vector of the same size.
  - (Note: For multiplying two vectors we use `.*` instead of just `*`, the same for `.^` when doing powers)
- Many mathematical functions we know will also operate on element-wise.
- Another helpful tool will be that the `length` function will return the length of our vector!

```
>> x = [5 4 8 1];
>> x(1)
ans =
     5
>> x(3)
ans =
     8
```

```
>> x + 2
ans =
     7     6    10     3
>> 3*x
ans =
    15    12    24     3
>> y = [1 2 3 4];
>> x + y
ans =
     6     6    11     5
>> x.*y
ans =
     5     8    24     4
```

```
>> x = [0 1 2];
>> exp(x)
ans =
    1.0000    2.7183    7.3891
>> log(x)
ans =
      -Inf         0    0.6931
>> sin(pi*x)
ans =
   1.0e-15 *
         0    0.1225   -0.2449
>> x.^2
ans =
     0     1     4
```

Vector Use in Loops

- In some cases, we want to generate a vector that depends on earlier elements of the vector
  - Sequences, integration, solving differential equations
- A fun example of this is the Fibonacci Sequence, we start with 1, 1, then each value is the sum of the two that come before it. This is represented mathematically as

$$x_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ x_{n-2} + x_{n-1} & \text{otherwise} \end{cases}$$

Vector Use in Loops

- In some cases, we want to generate a vector that depends on earlier elements of the vector
  - Sequences, integration, solving differential equations
- A fun example of this is the Fibonacci Sequence, we start with 1, 1, then each value is the sum of the two that come before it. This is represented mathematically as

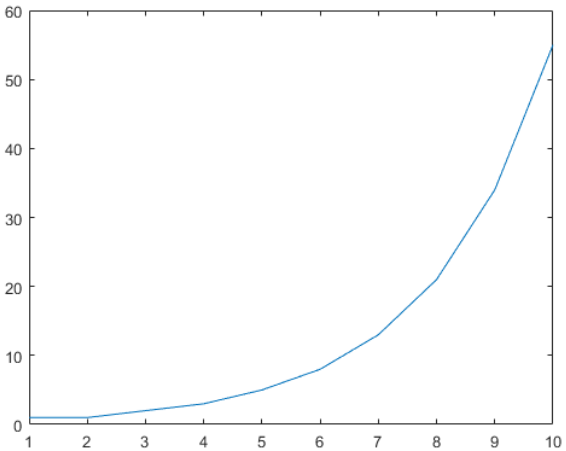
$$x_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ x_{n-2} + x_{n-1} & \text{otherwise} \end{cases}$$

- Lets see it done in Matlab

```
% How many terms do we want?
N = 10;

% Preallocate our vector as all zeros
fib_seq = zeros(N,1);

% If N is 1 or 2, we already have the answer!
for ii = 1:length(fib_seq)
    if ii == 1 || ii == 2   % Terms 1 and 2 have special rules (both are set to 1)
        fib_seq(ii) = 1;
    else                    % For everything else, add the last 2 terms
        fib_seq(ii) = fib_seq(ii-2) + fib_seq(ii-1);
    end
end

fib_seq
```

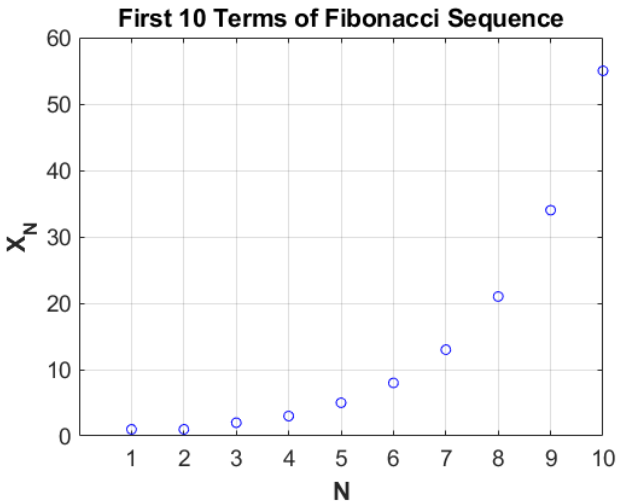
Plotting Vectors

- As you noticed, I didn't show the values of `fib_seq` on the previous slide. There is a more efficient way to look at vectors, that is by plotting them! Which we can do just by `plot(fib_seq)`
  - With a little more TLC, we can make the plot a lot easier to understand!



Plotting Vectors

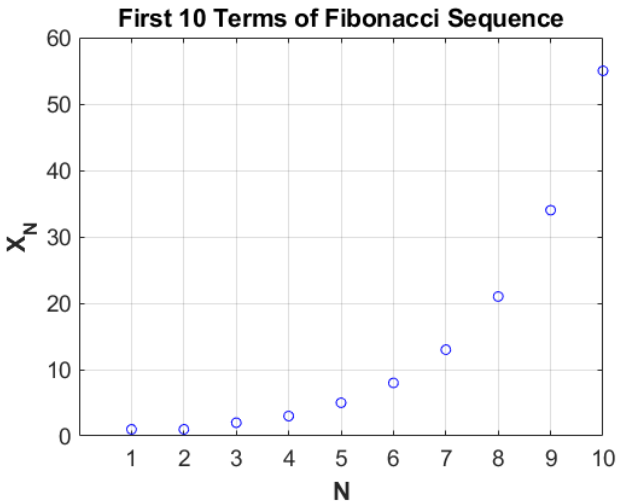
- As you noticed, I didn't show the values of `fib_seq` on the previous slide. There is a more efficient way to look at vectors, that is by plotting them! Which we can do just by `plot(fib_seq)`
  - With a little more TLC, we can make the plot a lot easier to understand!

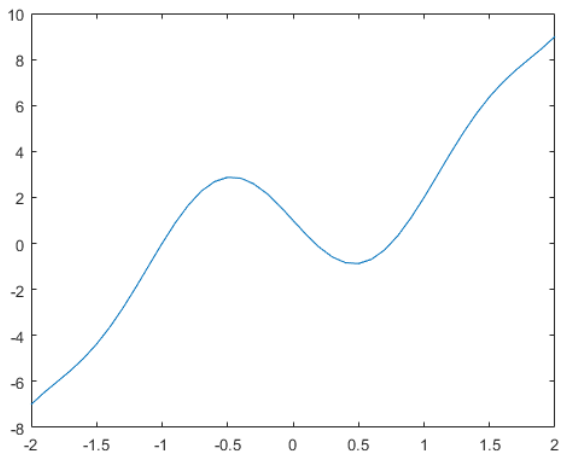


Plotting Vectors

- As you noticed, I didn't show the values of `fib_seq` on the previous slide. There is a more efficient way to look at vectors, that is by plotting them! Which we can do just by `plot(fib_seq)`
  - With a little more TLC, we can make the plot a lot easier to understand!
- We can also plot functions! Let's use the plot function to see what $x^3 - 2\sin(x\pi) + 1$ looks like for $-2 < x < 2$.

```
>> x = -2:0.1:2;
>> y = x.^3 - 2*sin(pi*x) + 1;
>> plot(x,y)
```



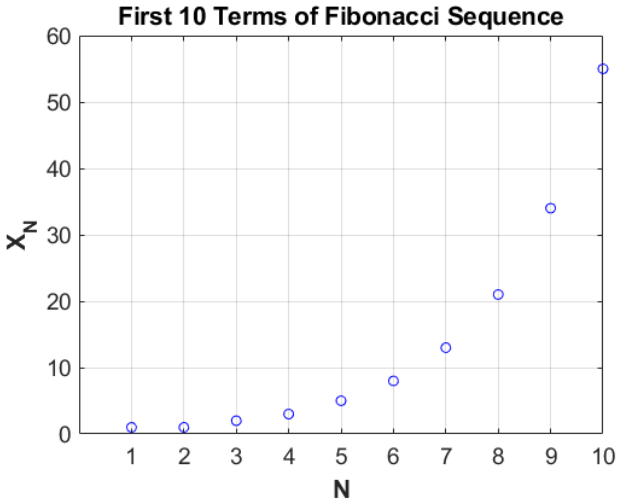


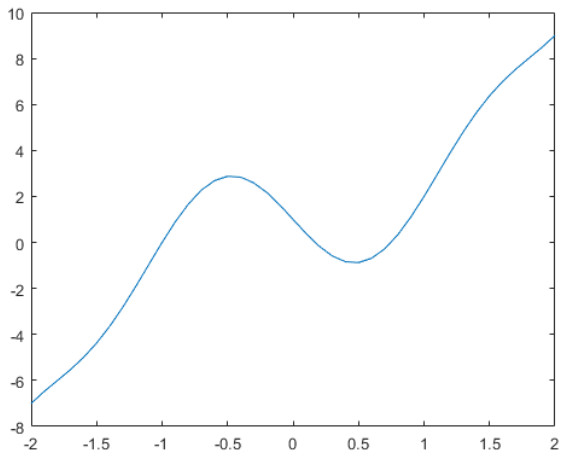
Plotting Vectors

- As you noticed, I didn't show the values of `fib_seq` on the previous slide. There is a more efficient way to look at vectors, that is by plotting them! Which we can do just by `plot(fib_seq)`
  - With a little more TLC, we can make the plot a lot easier to understand!
- We can also plot functions! Let's use the plot function to see what $x^3 - 2\sin(x\pi) + 1$ looks like for $-2 < x < 2$.

```
>> x = -2:0.1:2;
>> y = x.^3 - 2*sin(pi*x) + 1;
>> plot(x,y)
```

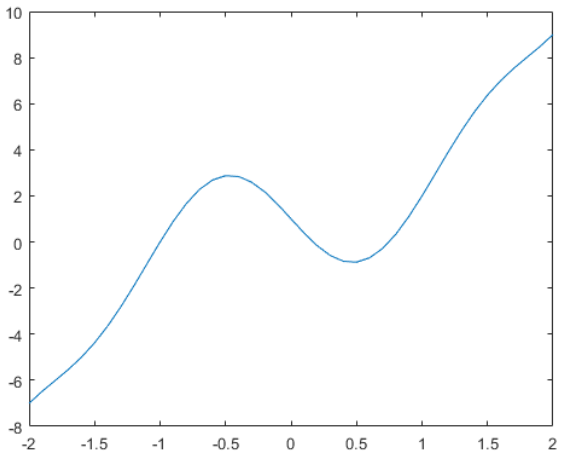
- Lets use this plot to show how to make plots look better!





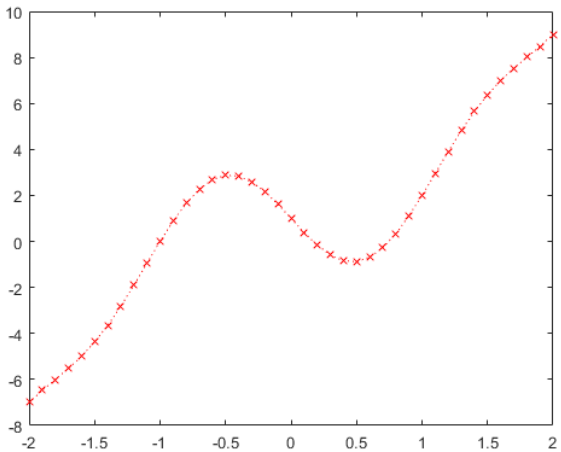
Axis Options

- Let's look at what we can do to give this plot a makeover!



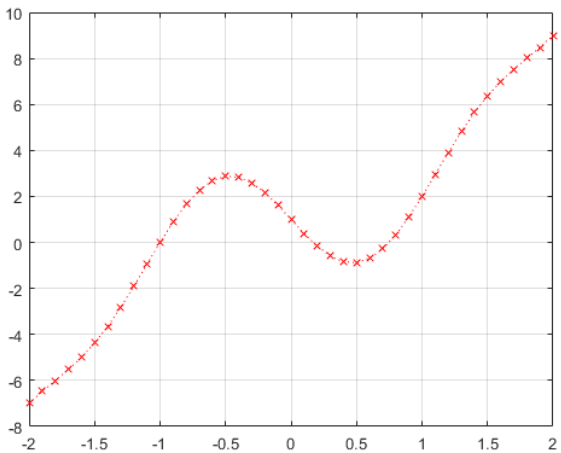
Axis Options

- Let's look at what we can do to give this plot a makeover!
- First, let's change how the data is being plotted
  `plot(x, y, 'rx:')`
  - The `r` makes the data plot in red
  - `x` says to display an x on the vector points
  - `:` indicates to draw a dotted line between points
  - Use `help plot` for more options!



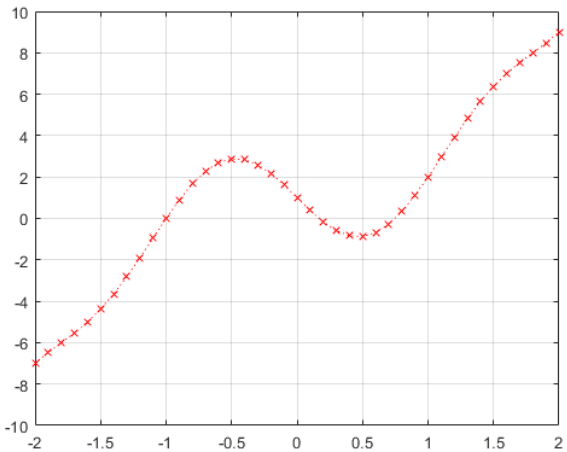
Axis Options

- Let's look at what we can do to give this plot a makeover!
- First, let's change how the data is being plotted
  `plot(x, y, 'rx:')`
  - The `r` makes the data plot in red
  - `x` says to display an x on the vector points
  - `:` indicates to draw a dotted line between points
  - Use `help plot` for more options!
- Now, lets turn on a grid
  `grid on;`



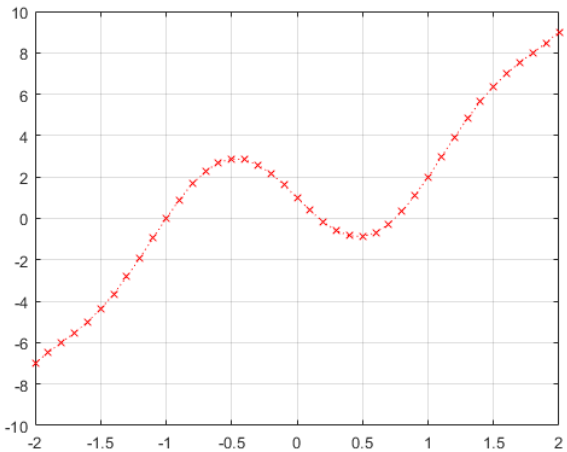
Axis Options

- Let's look at what we can do to give this plot a makeover!
- First, let's change how the data is being plotted
  `plot(x, y, 'rx:')`
  - The `r` makes the data plot in red
  - `x` says to display an x on the vector points
  - `:` indicates to draw a dotted line between points
  - Use `help plot` for more options!
- Now, lets turn on a grid
  `grid on;`
- The y-axis is not centered at 0. Lets do -2 to 2 on the x-axis and -10 to 10 on the y-axis
  `xlim([-2 2]); ylim([-10 10]);`



Plot Labeling

- This is starting to look good, but what is being plotted? We should label our plot!

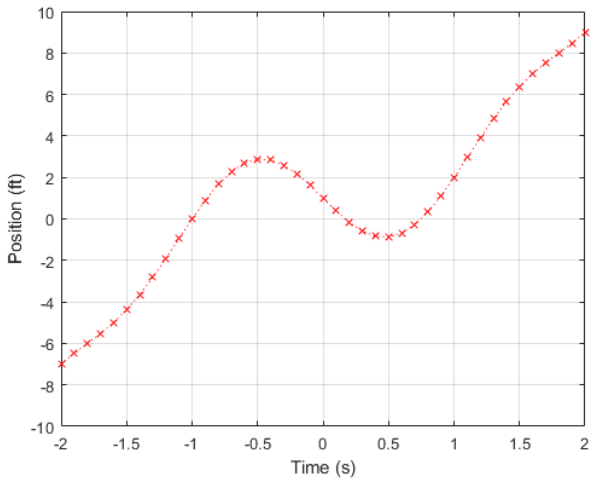


Plot Labeling

- This is starting to look good, but what is being plotted? We should label our plot!
- Suppose this is a plot of an object's position vs time! Let's label our axes appropriately

```
xlabel('Time (s)');
ylabel('Position (ft)');
```

  - Note: The `' ... '` tell Matlab that everything is a **string**, or to treat the inside as text



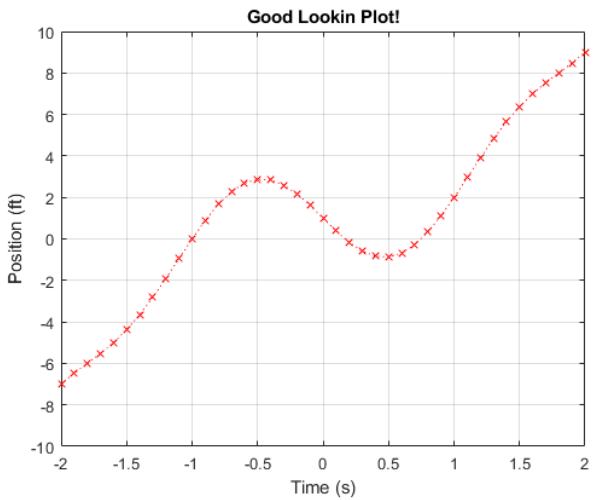
Plot Labeling

- This is starting to look good, but what is being plotted? We should label our plot!
- Suppose this is a plot of an object's position vs time! Let's label our axes appropriately

```
xlabel('Time (s)');
ylabel('Position (ft)');
```

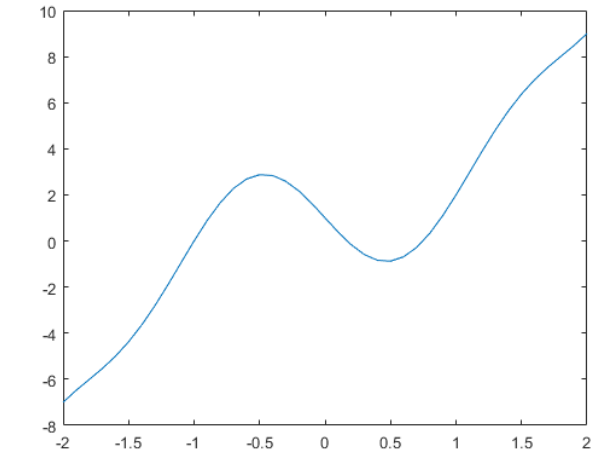
  - Note: The `' ... '` tell Matlab that everything is a **string**, or to treat the inside as text
- Finally, lets give it a title.

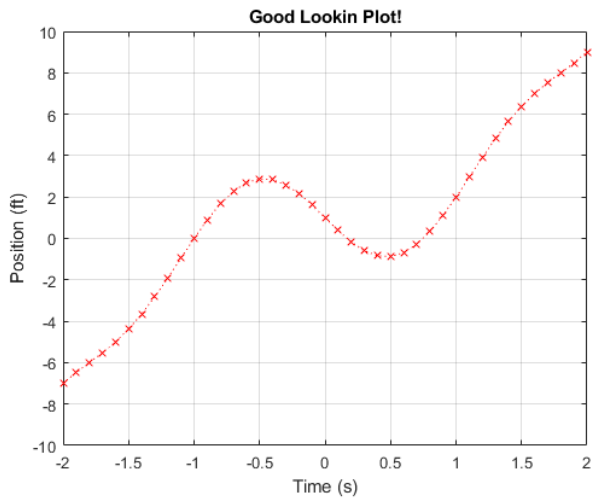
```
title('Good Lookin Plot!');
```



Plot Labeling

- This is starting to look good, but what is being plotted? We should label our plot!
- Suppose this is a plot of an object's position vs time! Let's label our axes appropriately
  `xlabel('Time (s)');`
  `ylabel('Position (ft)');`
  - Note: The `' ... '` tell Matlab that everything is a **string**, or to treat the inside as text
- Finally, lets give it a title.
  `title('Good Lookin Plot!');`
- Wow! Look at that glow up!





Multiple Plots

- If we tried to plot something new, it would get rid of all our hard work. We don't want that! So we have two options
  - If we want a new plot. We can enter `figure();` and open a new plot window.
  - If we want to add to our existing plot we can use `hold on;` to tell Matlab to hold onto the plot as is.

```
hold on;
plot(x, sin(pi*x), 'k-');
legend('x^3 - 2sin(pi*x) + 1', 'sin(pi*x)')
```

- There are a lot more options you can use for making plots!