In [1]:
```
using LinearAlgebra
```

In [2]:
```
using PyPlot
```

Linear Algebra

One of the most important problems (and most studied problems) is that of solving $Ax = b$. It's actually surprisingly difficult for various reasons (speed, cutoff error, stability, etc...)

Let's look at one such problem with stability:

In [3]:
```
A = [ [2.0, 1.2] [0.9, 0.54+5e-10] ]
```

Out[3]:
```
2×2 Matrix{Float64}:
 2.0  0.9
 1.2  0.54
```

In [4]:
```
inv(A)
```

Out[4]:
```
2×2 Matrix{Float64}:
  5.4e8  -9.0e8
 -1.2e9   2.0e9
```

In [5]:
```
b = [ 100.1, 200.2 ]
```

Out[5]:
```
2-element Vector{Float64}:
 100.1
 200.2
```

In [6]:
```
x = inv(A)*b
```

Out[6]:
```
2-element Vector{Float64}:
 -1.261259895142388e11
  2.8027997680953076e11
```

In [7]:
```
norm( A * x - b )
```

Out[7]:
```
0.00012207031251136867
```

So the norm is clearly not zero, so this did not solve $Ax = b$ properly... What happened?

The answer is that the above matrix is extremely close to being singular:

In [8]:
```
det(A)
```

Out[8]:
```
1.000000082740371e-9
```

This is one of the key points of numerical algorithms - stability. We are not going to go into a lot of detail here at all, but the closer your matrix is to being singular, the worse numerically it will perform. This is **entirely** due to numerical cutoff, algebraically everything is still exact.

How can we quantify how bad? That is how close to being singular? This is the concept of a condition number:

In [9]:
```
cond(A)
```

Out[9]:
```
6.541601390762973e9
```

The larger the condition number the closer to being singular. The optimal is 1, the identity matrix. There are ways to get around inverting this that are more stable:

In [10]:
```
x2 = A \ b
```

Out[10]:
```
2-element Vector{Float64}:
 -1.2612598951423882e11
  2.802799768095307e11
```

In [11]:
```
norm( A*x2 - b )
```

Out[11]:
```
1.364787585194269e-5
```

This is better but still not amazing - there are fancier methods to do this (outside scope of this course) such as preconditioning etc.

Moral of the story is ill-conditioned matrices lead to bad numerical answers and that some algorithms fare better in bad cases than others.

Polynomial Interpolation

This is related to your hw problem but will need to be modified. The problem is this: given n pairs of x,y coordinates, can we interpolate a polynomial through these points? (Hint: the answer is yes)

This is actually a linear algebra problem.

Say we have $n = 1$ points. Then hopefully should be easy to see the interpolating polynomial here is the constant one through that point.

Say we have 3 points (x1,y1), (x2,y2), (x3,y3). Then we can interpolate a polynomial through this of the form $p(x) = ax^2 + bx + c$. Why? 3 degrees of freedom.

Thus we need to satisfy:

$$p(x_1) = y_1$$
$$p(x_2) = y_2$$
$$p(x_3) = y_3$$

This is same as the following linear system:

In [12]:
```
x = rand(3); y = rand(3);
println(x)
println(y)
```

```
[0.6640692954012222, 0.7365261153556082, 0.8843757675554409]
[0.27962598475420775, 0.39883581471865637, 0.5611293551730343]
```

In [13]:
```
A = [ [1,1,1] [x[1], x[2], x[3]] [x[1]^2, x[2]^2, x[3]^2] ]
```

Out[13]:
```
3×3 Matrix{Float64}:
 1.0  0.664069  0.440988
 1.0  0.736526  0.542471
 1.0  0.884376  0.78212
```

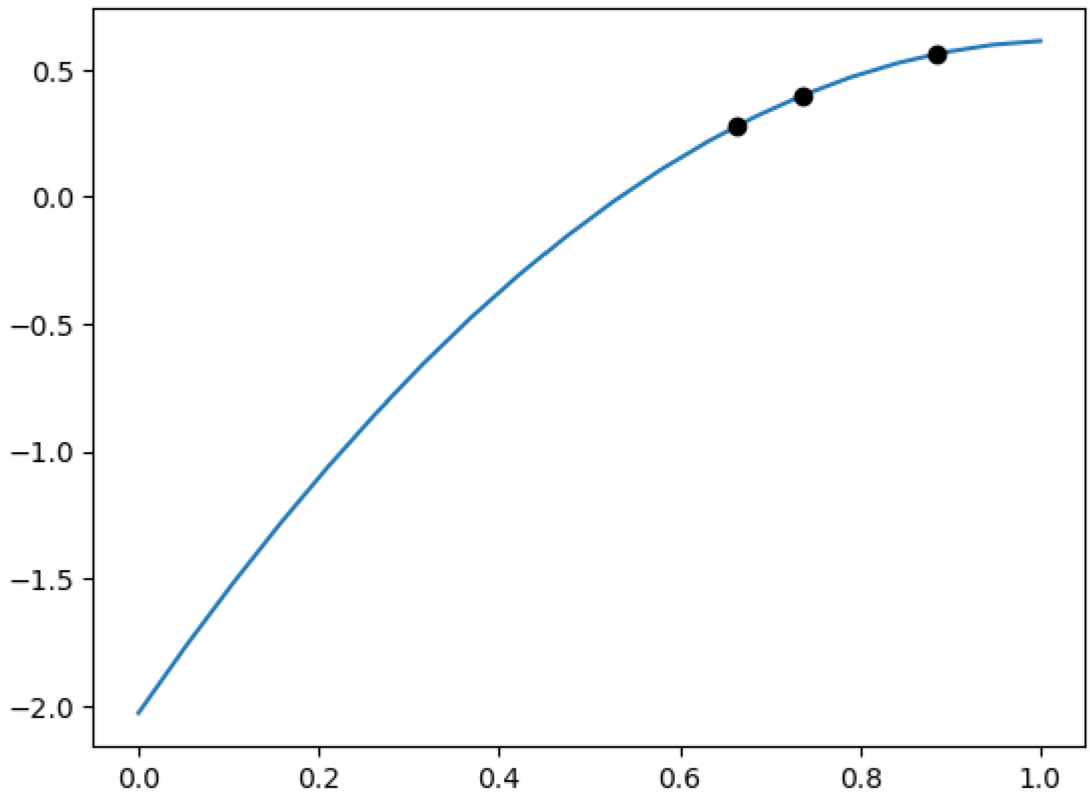
In [14]:
```
coeffs = A \ y
```

Out[14]:
```
3-element Vector{Float64}:
 -2.0285798858832567
  5.126360756738726
 -2.4854482043710666
```

In [15]:
```
p(x) = coeffs[1] + coeffs[2]*x + coeffs[3]*x^2
```

Out[15]:
```
p (generic function with 1 method)
```

In [16]:
```
xpts = LinRange(0,1,20)
ypts = [p(x) for x in xpts]
plot( xpts, ypts )
plot( x[1], y[1], "ko")
plot( x[2], y[2], "ko")
plot( x[3], y[3], "ko")
```



Out[16]:
```
1-element Vector{PyCall.PyObject}:
 PyObject <matplotlib.lines.Line2D object at 0x7f16edba4e20>
```

The matrix A is called a Vandermonde matrix. It turns out that they can be very ill-conditioned in practice but there are fancy ways to deal with this.

String Processing

Strings are just words basically:

In [17]:
```
println( "hello, I am a string" )
```

```
hello, I am a string
```

Special characters are indicated with a backslash, for example `\n` is string for newline, `\t` for tab

In [18]:
```
println( "hello \n I am a string" )
```

```
hello
 I am a string
```

To put two strings together

In [19]:
```
string( "Hi", " I am a string")
```

Out[19]:
```
"Hi I am a string"
```

To add in other types as arguments to strings, use the `$` operator

In [20]:
```
println( "Vandermonde A: $A" )
```

```
Vandermonde A: [1.0 0.6640692954012222 0.44098802909467566; 1.0 0.7365
261153556082 0.5424707186008226; 1.0 0.8843757675554409 0.782120498239
2752]
```

Strings are honestly just arrays of characters, so can index into them as such

In [21]:
```
"Vandermonde"[3]
```

Out[21]:
```
'n': ASCII/Unicode U+006E (category Ll: Letter, lowercase)
```

To convert string input into other types, use the `parse` command

In [22]:
```
parse( Float64, "1234")
```

Out[22]:
```
1234.0
```

This is really helpful when reading in data, as **ALL** data read in from any file is always in string format. To read in a file/writing from a file, you need to open the file, then you can read it in line by line

```
 f = open("filename.ext")
str = readline(f)
```

Obviously the filename needs to be correct. To write similarly

```
 write(f, "some string")
```

Lastly you can also search for patterns in strings:

In [23]:
```
str = "Hello, World! These are my words."
pattern = "wor"
idx1 = findfirst(pattern, lowercase(str))
```

Out[23]:
```
8:10
```

Which will give you the position of the desired pattern. One useful tool for this is regex (regular expressions)

In [24]:
```
str1 = "birthday=01/01/2000"
pattern = r"\d{2}/\d{2}/\d{4}"
idx1 = findfirst(pattern, str1)
```

Out[24]:
```
10:19
```

You can read more about this yourself online or come ask me in OH, but the way it works for the above is `\d` is any digit 0-9, {2} means it appears twice. So we are looking for something of the format dd/mm/yyyy. It will then return the position of anything that matches this pattern. Here is a list of regex commands (common ones)

\d is digits = [0-9]

\w is alphabetical letter = [a-z]

capital letters [A-Z]

any letter [A-Za-z]

how many times it appear, put {number} afterwards

\s is whitespace

In [25]:
```
randstring = "asdasdnq192371jajs12"
```

Out[25]:
```
"asdasdnq192371jajs12"
```

In [26]:
```
pattern4 = r"\w\d{3}"
```

Out[26]:
```
r"\w\d{3}"
```

In [27]:
```
findall( pattern4, randstring )
```

Out[27]:
```
1-element Vector{UnitRange{Int64}}:
 8:11
```

In [28]:
```
randstring[ findfirst( pattern4, randstring ) ]
```

Out[28]:
```
"q192"
```

In [ ]:
```

```