Numerical Computing :: Project 9

Julia Troni

In [1]:
```
%matplotlib notebook
from matplotlib import pyplot
import matplotlib.pyplot as plt
import numpy as np
import math
```

In [ ]:
```

```

---

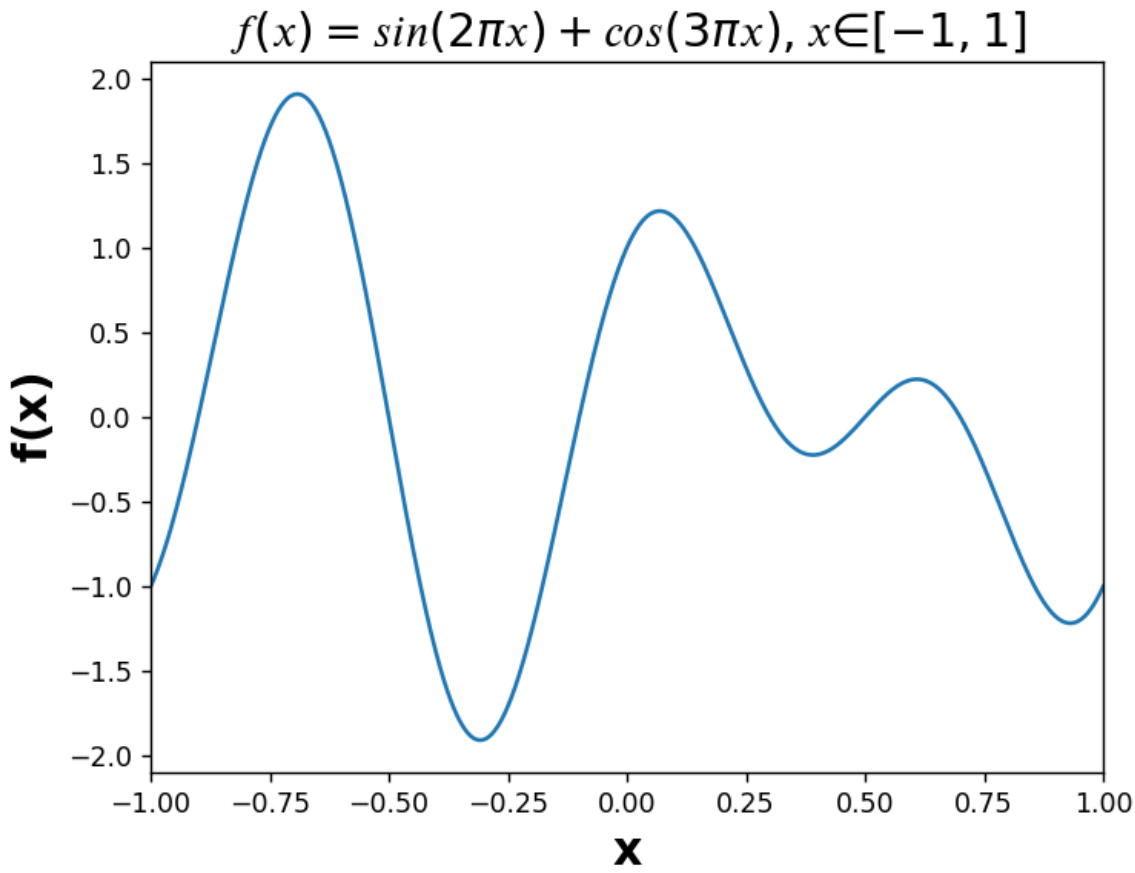
1. Method of Normal Equations (uses the Cholesky factorization)

Consider the function

$f(x) = sin(2\pi x) + cos(3\pi x), x \in [-1, 1].$

Evaluate the function values at n evenly spaced points. You get to choose n. For d from 0 to n − 1, compute the least-squares coefficients of a polynomial of degree d with the same training data using both the QR method and the normal equations.

In [2]:
```
interval = -1, 1
f = lambda x: np.sin(2*np.pi*x) + np.cos(3*np.pi*x)
xs = np.linspace(interval[0], interval[1], num=8000)

plt.figure()
plt.plot(xs, f(xs))
plt.title("$f(x)=sin(2\pi x)+cos(3\pi x), x\in[-1,1]$", fontsize='xx-large')
plt.xlim([interval[0], interval[1]])
plt.xlabel("$\\bf{x}$", fontsize='xx-large')
plt.ylabel("$\\bf{f(x)}$", fontsize='xx-large')


plt.draw()
plt.show()
```



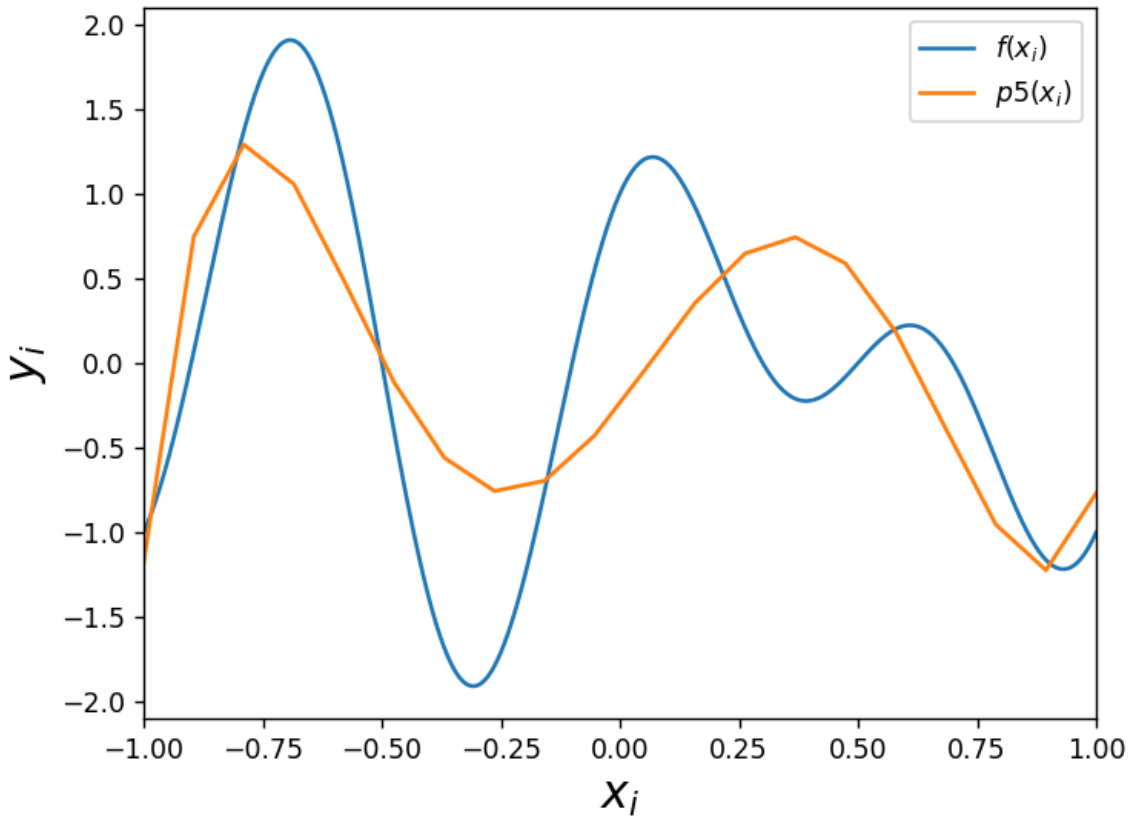
In [3]:
```
interval = -1, 1
f = lambda x: np.sin(2*np.pi*x) + np.cos(3*np.pi*x)
```

In [50]:
```
#n=20 number of evenly spaced points.
xis= np.linspace(-1,1,20)
```

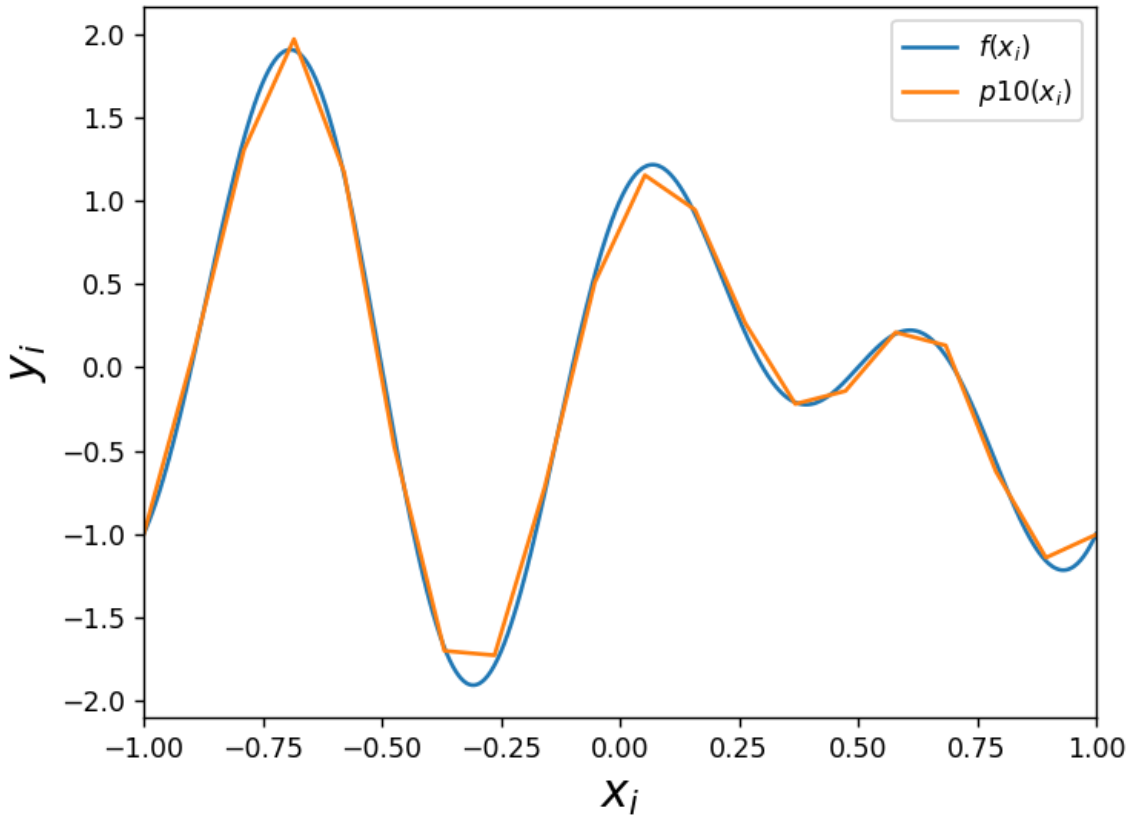
In [51]:
```
#d is degree of p(x)
#f is the actual function we want to approximate
#xis are the x values
def EvalAndGraphLeastSquares(d,f, xis):

    # vandermonde matrix to represent our over-determined system
    A = np.vander(xis, N=d+1, increasing=True)

    # actual values of f(x) that were trying to approximate
    yis = np.array([f(x) for x in xis])
    #print("ys ", yis)
    #print("xs ", xis)

    # our coefficients for the polynomial
    cs = np.linalg.lstsq(A, yis, rcond=None)[0]
    #print("cs ", cs)

    def fclose(cs, xis):
        return sum([cs[i] * (xis ** i) for i in range(len(cs))])

    plt.figure()
```

```
    plt.subplot(1,1,1)
    plt.plot(xs, f(xs), label='$f(x_{i})$')
    plt.plot(xis, fclose(cs,xis) , label='$p{}(x_i)$'.format(d))
    plt.xlim([interval[0], interval[1]])
    plt.xlabel("$x_{i}$", fontsize='xx-large')
    plt.ylabel("$y_{i}$", fontsize='xx-large')
    plt.legend()
    plt.show()
```

In [52]:
```
EvalAndGraphLeastSquares(5, f, xis)
```

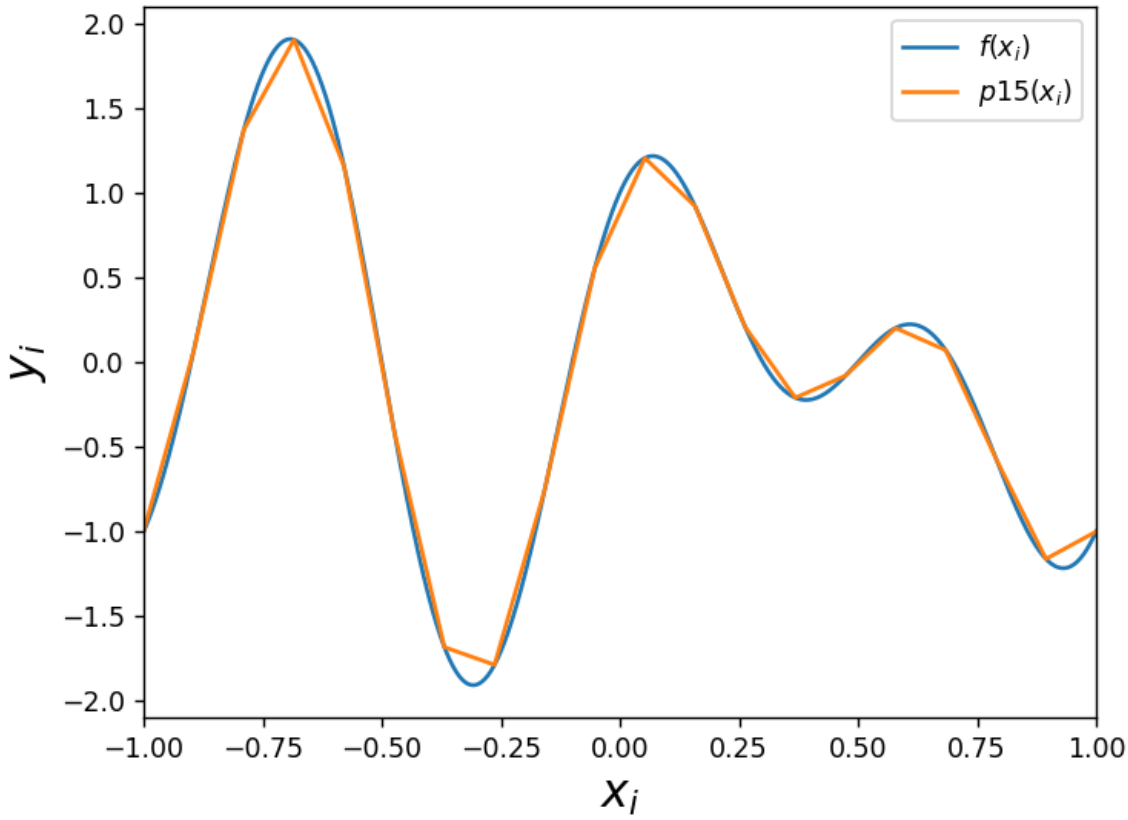


In [53]:
```
EvalAndGraphLeastSquares(10, f, xis)
```



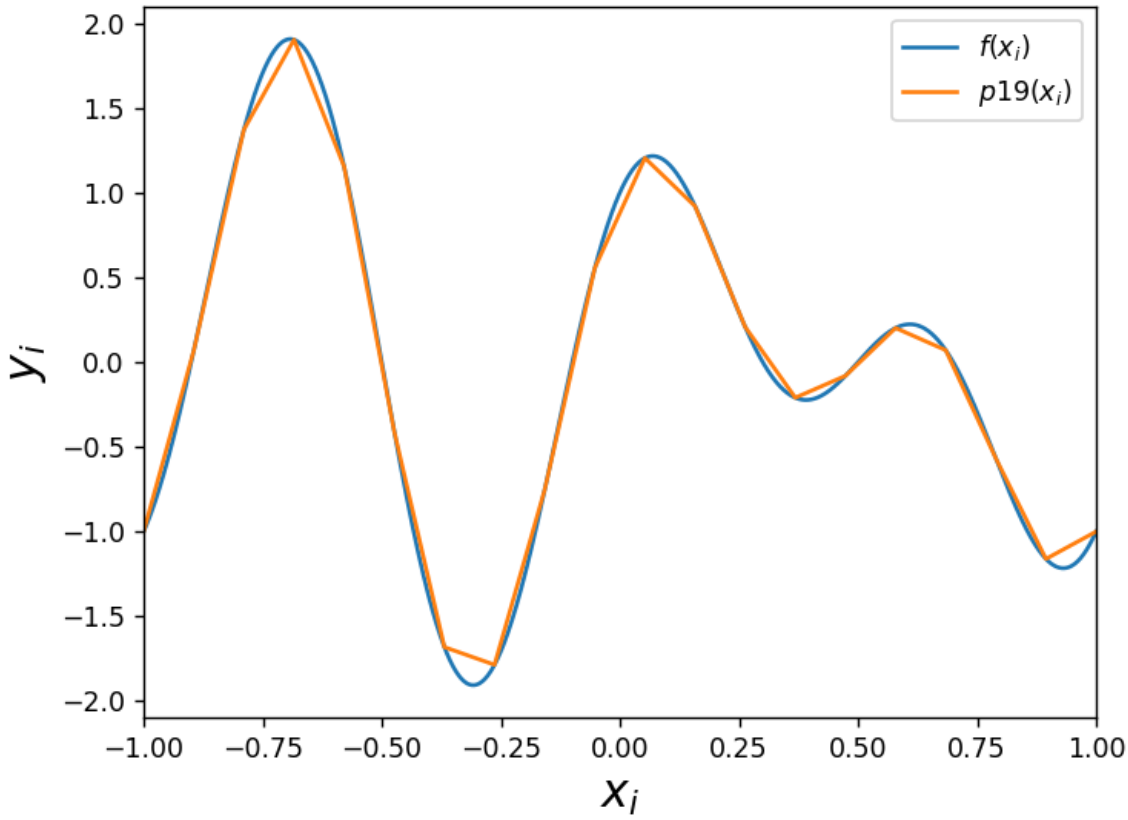
In [54]:

```
EvalAndGraphLeastSquares(15, f, xis)
```



In [ ]:

In [55]:
```
EvalAndGraphLeastSquares(19, f, xis)
```



In [ ]:

When the number of points is larger than the degree of polynomial that you want to use, then the linear system for determining the coefficients will be over-determined (more rows than columns). To find the least-squares polynomial of a given degree, you carry out the same process as we did for interpolation, but the resulting polynomial will not interpolate the data, it will just be "close"

2. For each trained polynomial compute testing error on a set of testing data, which you will generate. Plot the error e versus d on a semilogy scale. Make sure to include both

- (i) the error computed using the QR decomposition
- (ii) the error computed using the normal equations.

Interpret the error behavior. (HINT: It's related to the condition number of the matrix in the least-squares problem.)

- Testing Error

$$e_d = \left(\frac{\Sigma_{i=1}^{100}(y_i^{'} - p(x_i^{'}))^2}{\Sigma_{i=1}^{100}(y_i^{'})^2}\right)^{\frac{1}{2}}$$

In [56]:

```
#solve normal and QR
```

```
def choleskySolve(A, b):
    new_A = np.matmul(A.T, A)
    new_b = np.matmul(A.T, b)
    L = np.linalg.cholesky(new_A)
    y = np.linalg.solve(L, new_b)
    return np.linalg.solve(L.T, y)

#Solve with normal equations
def NormEqChol(A, b):
    if np.linalg.matrix_rank(A) == A.shape[1]:
        return choleskySolve(A,b)
    else:
        LU = linalg.lu_factor(A)
        #solve given LU and b
        x = linalg.lu_solve(LU, b)
        return x


def LU_solve(matrix):
    b = generate_bs(matrix)
    #call the lu_factor function
    LU = linalg.lu_factor(matrix)
    #solve given LU and b
    x = linalg.lu_solve(LU, b)
    return x

#Thin QR
def ThinQR(A, b):
    Q, R = np.linalg.qr(A)
    return np.linalg.solve(R, Q.T.dot(b))

def fclose(cs, xis):
        return sum([cs[i] * (xis ** i) for i in range(len(cs))])
```

In [70]:

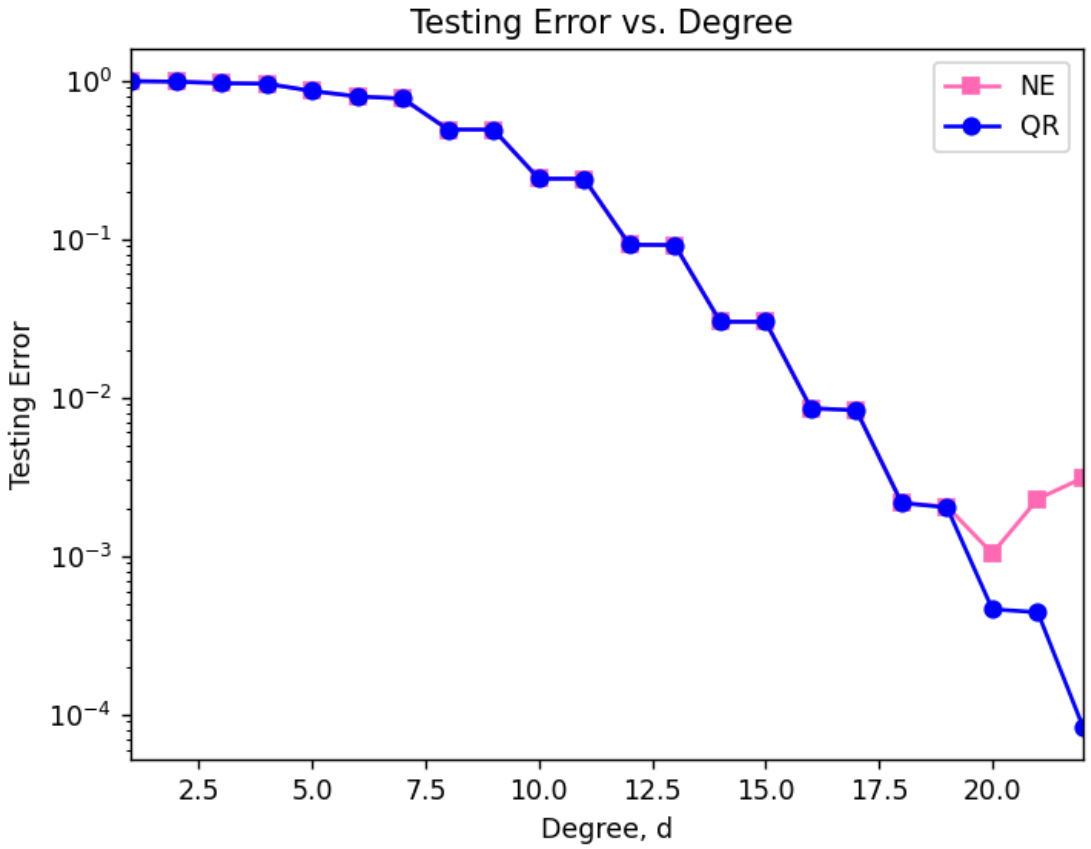
```
def get_error(obs, truth):
    diff = np.subtract(obs, truth)
    return np.sqrt(np.linalg.norm(diff, ord=2) / np.linalg.norm(truth, ord=2))

def generate_TestData(low=-1, high=1, n=100):
    l1 = np.append(np.array([low]), low * np.random.rand(n//2 - 1))
    l2 = np.append(np.array([high]), high * np.random.rand(n//2 - 1))
    ret = np.append(l1, l2)
    return np.sort(ret)

# i
new_xs = generate_TestData()

# ii
new_ys = np.array([f(x) for x in new_xs])
high = 22

qr_error = []
ne_error = []

for i in range(1, high+ 1):
    ai = np.vander(new_xs, N=i+1, increasing=True)

    qr_cs   = ThinQR(ai, new_ys)
```

```
    ####UGHHH MY NORMAL SOLVER IS NOT WORKING :/
    ne_cs    = NormEqChol(ai, new_ys)

    qr_approx = [fclose(qr_cs, x) for x in new_xs]
    ne_approx = [fclose(ne_cs, x) for x in new_xs]

    qr_error.append(get_error(qr_approx, new_ys))
    ne_error.append(get_error(ne_approx, new_ys))
```

In [ ]:

In [71]:

```
low =1
#for columns
k = [n for n in range(low, high + 1)]

#semilog plot with mathlab
plt.figure()
plt.subplot(1,1,1)
plt.semilogy(k, ne_error, '-s', label='NE', color='hotpink')
plt.semilogy(k, qr_error, '-o', label='QR', color= 'blue')
plt.xlim([low, high])

#label
plt.title("Testing Error vs. Degree ")
plt.xlabel("Degree, d")
plt.ylabel("Testing Error")
plt.legend()
plt.show()
```



The testing error goes down initially, for both QR and Normal Equations, but then the testing for normal equations starts to increase. From the previous project I recall learning that using a Normal Equations method ultimately squares the condition number of the problem matrix. So as degree increases it make sence that QR decomposition should have less error than Normal equation solver.

By squaring the condition number, the normal equation method is essentially creating more "noise" in the model, or a more complex model which is obviously harder to approximate, thus the error increases.

References

- Paul's famous lectures, as always ##### You shouldn't drink and derive LOL
- used many helper functions from my Project 5 and 8
- http://www.math.iit.edu/~fass/477577_Chapter_5.pdf
- https://pythontic.com/visualization/charts/semilog
- https://johnwlambert.github.io/least-squares/
- https://boostedml.com/2020/04/solving-full-rank-linear-least-squares-without-matrix-inversion-in-python-and-numpy.html

In [ ]: