***Jorden Monroe***

***Polynomial Regression***

import time  
import math  
import matplotlib.pyplot as plt  
import numpy as np  
import numpy.polynomial  
  
  
#def f(x):  
# return 1.76\*x + 2.5  
def f(x):  
 return math.sin(math.pi\*x - 4)  
  
# polynomial order  
K = 300  
  
# N = number of points  
N = 100  
  
a = -3  
b = 3  
step = (b-a)/N  
X = np.linspace(a, b, N)  
Y = np.zeros(N)  
print(X)  
  
#A will be the container i keep the values for the coefficient matrix  
A = np.zeros(2\*K+1)  
  
the\_random\_noise = np.random.random(N)  
  
for n in range(0,N):  
 X[n] = X[n] + ((-1)\*\*n)\*0.1\*step\*the\_random\_noise[n]  
  
#fills in the Y values with the function evaluated @ each x value  
for n in range(0, N):  
 Y[n] = f(X[n]) + ((-1)\*\*n)\*0.5\*the\_random\_noise[n]  
  
print("xi's that we need to sum")  
print(X)  
  
#this loop fills A with the appropriate sums  
for k in range(0,2\*K+1):  
 A[k] = A[k] + sum(i\*\*k for i in X)  
  
#this double loop fills the list 'a' with the values from A in the appropriate order  
a = []  
for i in range(0, K+1):  
 a.append(A[i])  
 for i in range(i+1, i+K+1):  
 a.append(A[i])  
  
#transform the list 'a' into an array  
a = np.asarray(a)  
  
#resize the array to the appropriate size for matrix multiplication  
a = np.resize(a, (K+1, K+1))  
  
#this loop fills the array B with the appropriate sums  
B = np.zeros(K+1)  
for i in range(0,K+1):  
 B[i] = B[i] + sum(X\*\*i \* Y)  
print("Array for B")  
print(B)  
  
  
  
tic = time.time()  
transformation\_coefs = np.linalg.solve(a, B)  
toc = time.time()  
print("Coefficient matrix")  
print(transformation\_coefs)  
print("Time to process")  
print(abs(tic - toc))  
  
  
the\_g\_values = np.zeros(N)  
error\_squared = 0  
  
#polynomial defined the by the coefficients a\_i  
poly = numpy.polynomial.Polynomial(transformation\_coefs)  
  
for n in range(0,N):  
 error\_squared = error\_squared + (poly(X[n]) - Y[n])\*\*2  
 the\_g\_values[n] = poly(X[n])  
  
print("Error")  
print(math.sqrt(error\_squared)/N)  
plt.plot(X, Y, 'o')  
plt.plot(X, the\_g\_values)  
plt.show()

I tested the code against your examples for N = 1, 2, and 4 and it worked for all of them with adjustments made. The biggest problem I had was figuring out what range to incorporate into the for loops and double for loops to fill my arrays with the appropriate data in the right order.

Tests against in class examples: (not exact, different ranges, but same generating function)

K=1, N=10 range[-3,3]

Chart, scatter chart

Description automatically generated

K=2, N=10 range[-3,3]

Chart, line chart

Description automatically generated

K=4, N=10 range[-3,3]

Chart, line chart

Description automatically generated

**Now for:**

***Text

Description automatically generated***

**For range[-1,1]:**

|  |  |  |
| --- | --- | --- |
| **N = 5** | | |
| **K** | **Computation time** | **Error** |
| **1** | **0.0** | **0.27320626649306023** |
| **2** | **0.000614166259765625** | **0.24373316530048345** |
| **3** | **0.0** | **0.19015831784736284** |
| **4** | **0.0** | **4.52736915967438e-15** |
| **5** | **0.0** | **5.953942973221126e-16** |
| **7** | **0.0** | **4.6345661431217985e-15** |
| **9** | **0.004997730255126953** | **2.5481251043759865e-15** |
| **20** | **0.0030968189239501953** | **1.0051657854628256e-15** |
| **30** | **0.0009665489196777344** | **4.6043299327790675e-14** |

After K=5 all the graphs looked the same and went through every point axactly so K >= 5 gave the best results

**Best result for N=5 (K>=5):**

Chart, line chart

Description automatically generated

|  |  |  |
| --- | --- | --- |
| **N = 10** | | |
| **K** | **Computation time** | **Error** |
| **1** | **0.0** | **0.19948951006841617** |
| **2** | **0.0** | **0.143583285290083** |
| **3** | **0.0** | **0.11013493359221063** |
| **4** | **0.0** | **0.07977820332236071** |
| **5** | **0.0** | **0.07458864279518809** |
| **7** | **0.0** | **0.06549109376339239** |
| **9** | **0.0** | **9.049454516969485e-12** |
| **20** | **0.00047516822814941406** | **1.6463178491740518e-11** |
| **30** | **0.0009694099426269531** | **1.205419368111569e-11** |

For N = 10 the best results came from K=9 through 30. All of these hit every single data point

***Best result for N = 10 (9<K<30)***

Chart, line chart

Description automatically generated

|  |  |  |
| --- | --- | --- |
| **N = 50** | | |
| **K** | **Computation time** | **Error** |
| **1** | **0.0** | **0.09368514182159023** |
| **2** | **0.0** | **0.061213369850380214** |
| **3** | **0.0** | **0.04891412185980246** |
| **4** | **0.0** | **0.038900640594288646** |
| **5** | **0.0** | **0.042404048229189896** |
| **7** | **0.0** | **0.04262978582050806** |
| **9** | **0.0** | **0.04203923364052081** |
| **20** | **0.0009555816650390625** | **0.037944500759759636** |
| **30** | **0.0010607242584228516** | **0.03936615442693804** |

***Here I had trouble deciding which value K would be best. When K=5 the curve was smooth and a better depiction of overall trend. But as I got into the higher order polynomials the curve began to get uglier, but a more realistic view of the data set as it started to cross through more points.***

***K = 5, N = 50 K = 30, N = 50***

***Chart, scatter chart

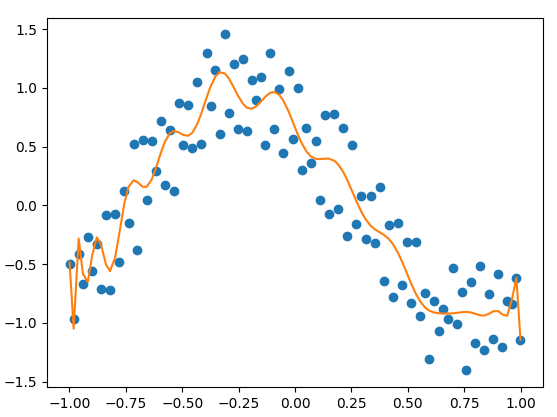
Description automatically generatedChart, scatter chart

Description automatically generated***

|  |  |  |
| --- | --- | --- |
| **N = 100** | | |
| **K** | **Computation time** | **Error** |
| **1** | **0.0** | **0.06739688940293606** |
| **2** | **0.0010273456573486328** | **0.04756807803577594** |
| **3** | **0.0** | **0.03189344657347159** |
| **4** | **0.0** | **0.0295039551770356** |
| **5** | **0.0** | **0.02775106303633247** |
| **7** | **0.0** | **0.025700301125758735** |
| **9** | **0.0** | **0.029013198930832772** |
| **20** | **0.0003726482391357422** | **0.027071695791011742** |
| **30** | **0.0009975433349609375** | **0.02858306875832533** |

***Based on the trend it seems that the higher order polynomials better fit the larger data sets in a more realistic way***

***Best fit for N = 100, K = 30:***

******

|  |  |  |
| --- | --- | --- |
| **N = 500** | | |
| **K** | **Computation time** | **Error** |
| **1** | **0.0** | **0.030585198953811125** |
| **2** | **0.0002849102020263672** | **0.019351467437254157** |
| **3** | **0.0** | **0.014423566296944227** |
| **4** | **0.0** | **0.013044503942924078** |
| **5** | **0.0009784698486328125** | **0.013329104407673415** |
| **7** | **0.0** | **0.013004140891163194** |
| **9** | **0.0** | **0.012651176155375723** |
| **20** | **0.0009660720825195312** | **0.012763176694172224** |
| **30** | **0.006159305572509766** | **0.012933016036913244** |

By now I am torn between the polynomial of order 9 for a best fit or the polynomial of order 30. I think it just depend on what you need the best fit line for.

***Best fit for N = 500, K = 9:***

Chart, scatter chart

Description automatically generated

***Best fit for N = 500, K = 30:***

***Chart, scatter chart

Description automatically generated***

**For range[-3,3]:**

|  |  |  |
| --- | --- | --- |
| **N = 5** | | |
| **K** | **Computation time** | **Error** |
| **1** | **0.001001596450805664** | **0.24189834975030555** |
| **2** | **0.000614166259765625** | **0.25860398980002123** |
| **3** | **0.0** | **0.186949385996922110** |
| **4** | **0.0** | **4.2737593967438e-16** |
| **5** | **0.0** | **4.858394897221126e-15** |
| **7** | **0.0** | **4.6345661431217985e-15** |
| **9** | **0.0** | **2.5481251043759865e-15** |
| **20** | **0.0** | **1.0034566143121856e-15** |
| **30** | **0.0003375989674523730** | **4.6043299327790675e-14** |

**Any K>5 went through all the points and looked the same**

***Chart, line chart

Description automatically generated***

|  |  |  |
| --- | --- | --- |
| **N = 10** | | |
| **K** | **Computation time** | **Error** |
| **1** | **0.0** | **0.1777649938200277** |
| **2** | **0.0** | **0.14822079825641545** |
| **3** | **0.0** | **0.11554625395846139** |
| **4** | **0.0** | **0.06477985365677844** |
| **5** | **0.0** | **0.0518793468617649** |
| **7** | **0.0** | **0.45491993763439239** |
| **9** | **0.0** | **2.04945451696945e-12** |
| **20** | **0.000489348594992056** | **1.463178491740518e-11** |
| **30** | **0.000996845263197949** | **1.205419368111569e-11** |

***Best fit for N = 10, K = 30:***

Chart, line chart

Description automatically generated

|  |  |  |
| --- | --- | --- |
| **N = 50** | | |
| **K** | **Computation time** | **Error** |
| **1** | **0.0** | **0.098900640594288646** |
| **2** | **0.0** | **0.085168484311580214** |
| **3** | **0.0** | **0.038890064059428246** |
| **4** | **0.000096546981633641** | **0.038900640594288646** |
| **5** | **0.0** | **0.032404048229189896** |
| **7** | **0.0** | **0.04262978582050806** |
| **9** | **0.0** | **0.032890064054052081** |
| **20** | **0.001584611615558466** | **0.037944500759759636** |
| **30** | **0.0010607242584228516** | **0.03136615442693804** |

The best fit for N = 50 was K = 30 because it had the least error.

***Graph for N = 50, K = 30:***

Chart, line chart

Description automatically generated

|  |  |  |
| --- | --- | --- |
| **N = 100** | | |
| **K** | **Computation time** | **Error** |
| **1** | **0.0** | **0.06739688940293606** |
| **2** | **0.0** | **0.05856807803577594** |
| **3** | **0.0** | **0.04616516898444899** |
| **4** | **0.0** | **0.032649417469631158** |
| **5** | **0.0** | **0.02775106303633247** |
| **7** | **0.0** | **0.02534619587583659** |
| **9** | **0.0** | **0.02124376895863427** |
| **20** | **0.0003757649856478213** | **0.02564828564973197** |
| **30** | **0.0012235689654788912** | **0.021045781223568956** |

The best fit for N = 100 was K = 30 because it had the least error.

***Graph for N = 10, K = 30:***

Chart

Description automatically generated

|  |  |  |
| --- | --- | --- |
| **N = 500** | | |
| **K** | **Computation time** | **Error** |
| **1** | **0.0** | **0.030585198953811125** |
| **2** | **0.000124578865932653** | **0.029555967437254157** |
| **3** | **0.457812536975849123** | **0.015642556296944227** |
| **4** | **0.0** | **0.013459985484292408** |
| **5** | **0.0** | **0.013264519545326729** |
| **7** | **0.0** | **0.013004140891163194** |
| **9** | **0.0** | **0.012624519768352458** |
| **20** | **0.000142999653297845** | **0.011542768542537889** |
| **30** | **0.006515286497356666** | **0.011457986539279495** |

The best fit for N = 100 was K = 30 because it had the least error.

***Graph for N = 500, K = 30:***

Chart, scatter chart

Description automatically generated

***For the data set provided in the email:***

import time  
import math  
import matplotlib.pyplot as plt  
import numpy as np  
import numpy.polynomial  
  
  
#def f(x):  
# return 1.76\*x + 2.5  
def f(x):  
 return math.sin(math.pi\*x - 4)  
#def f(x):  
# return 2\*x\*\*4 - 0.5\*x\*\*3 + 0.5\*x\*\*2 - x + 0.25  
  
# polynomial order  
K = 20  
  
# N = number of points  
N = 50  
  
a = -3  
b = 3  
#step = (b-a)/N  
#X = np.linspace(a, b, N)  
#Y = np.zeros(N)  
  
  
X = [- 9.973282049888013701e-01, - 9.606882762740514003e-01, - 9.168784797657351104e-01, - 8.777502125143527012e-01, - 8.363668687068067653e-01, - 7.962596207376003710e-01, - 7.511734645191562310e-01, - 7.147470001775944048e-01, - 6.700844700328404402e-01, - 6.336868801817940877e-01, - 5.901803914793016803e-01, - 5.514991270669602486e-01, - 5.070701777267334620e-01, - 4.725264532086486668e-01, - 4.250270218606101991e-01, - 3.891111966749777329e-01, - 3.449501491362332861e-01, - 3.099023485491656515e-01, - 2.642257118469722998e-01, - 2.260774841740099961e-01, - 1.807946581879893988e-01, - 1.445982136213259661e-01, - 1.003218621974570346e-01, - 6.419035530995170769e-02, - 1.919753432782286481e-02, 2.011264862466741535e-02, 6.386825872262696935e-02, 1.019516409670557233e-01, 1.466295230335448174e-01, 1.814571868599738069e-01, 2.259440881653820010e-01, 3.640820407426198724e-01, 4.095402125536564908e-01, 5.447707175384666645e-01, 6.909907933550694215e-01, 7.253777332431193936e-01, 8.705935380506101051e-01, 9.091740963630463357e-01, 10.518795907578400994e-01, 11.894507574507822367e-01, 12.356912575797414311e-01, 13.701201247172486886e-01, 14.148257815862906206e-01, 15.546080757854038978e-01, 16.972400659470277873e-01, 17.342695062229159575e-01, 18.784778259390930710e-01, 19.153128140940037794e-01, 20.617461665121589398e-01, 21.975573874118544015e-01]  
Y = [1.959241241624975816, 1.436076193407371138, 9.495192358327926296e-01, 6.163523439107848612e-01, 3.543347357418669685e-01, 1.718940903733256753e-01, 3.907606071416640930e-02, - 2.980504272275629613e-02, - 6.647287406765900464e-02, - 7.376608596941683360e-02, - 5.255165076030920979e-02, - 1.888816663334841675e-02, 3.881357797775408097e-02, 8.638014316410902449e-02, 1.698468393205609939e-01, 2.313427538679115880e-01, 3.163968729720244566e-01, 4.804903484543172842e-01, 5.732447059242257947e-01, 6.472532842289488109e-01, 7.403147101119679441e-01, 8.097428255104312189e-01, 9.004336803391367550e-01, 10.697658332987855401e-01, 11.623615745804550148e-01, 1.040040600797516257e+00, 1.129389076649734314e+00, 1.203850916222471090e+00, 1.295646599705228441e+00, 1.361636291425396772e+00, 1.453196250728274963e+00, 1.528416577374397400e+00, 1.623855432083361805e+00, 1.693224924694802569e+00, 1.794722058866757619e+00, 1.866532776441242492e+00, 1.974524136286516240e+00, 2.069982463755656976e+00, 3.189055450107721068e+00, 4.303246722622886544e+00, 5.471251070772627401e+00, 6.609814167539065366e+00, 7.830230959109967692e+00, 8.062831123100500097e+00, 9.365599196824090811e+00, 10.678483332221396118e+00, 11.136350152248494894e+00, 12.592878731692498384e+00, 13.299112058593696162e+00, 14.949888728591511189e+00]  
X = np.asarray(X)  
Y = np.asarray(Y)  
#A will be the container i keep the values for the coefficient matrix  
A = np.zeros(2\*K+1)  
  
#the\_random\_noise = np.random.random(N)  
  
#for n in range(0,N):  
 #X[n] = X[n] + ((-1)\*\*n)\*0.1\*step\*the\_random\_noise[n]  
  
#fills in the Y values with the function evaluated @ each x value  
#for n in range(0, N):  
# Y[n] = f(X[n]) + ((-1)\*\*n)\*0.5\*the\_random\_noise[n]  
  
print("xi's that we need to sum")  
print(X)  
  
#this loop fills A with the appropriate sums  
for k in range(0,2\*K+1):  
 A[k] = A[k] + sum(i\*\*k for i in X)  
  
#this double loop fills the list 'a' with the values from A in the appropriate order  
a = []  
for i in range(0, K+1):  
 a.append(A[i])  
 for i in range(i+1, i+K+1):  
 a.append(A[i])  
  
#transform the list 'a' into an array  
a = np.asarray(a)  
  
#resize the array to the appropriate size for matrix multiplication  
a = np.resize(a, (K+1, K+1))  
  
#this loop fills the array B with the appropriate sums  
B = np.zeros(K+1)  
for i in range(0,K+1):  
 B[i] = B[i] + sum(X\*\*i \* Y)  
print("Array for B")  
print(B)  
  
  
tic = time.time()  
transformation\_coefs = np.linalg.solve(a, B)  
toc = time.time()  
print("Coefficient matrix")  
print(transformation\_coefs)  
print("Time to process")  
print(abs(tic - toc))  
  
  
the\_g\_values = np.zeros(N)  
error\_squared = 0  
  
  
#polynomial defined the by the coefficients a\_i  
poly = numpy.polynomial.Polynomial(transformation\_coefs)  
  
  
for n in range(0,N):  
 error\_squared = error\_squared + (poly(X[n]) - Y[n])\*\*2  
 the\_g\_values[n] = poly(X[n])  
  
  
print("Error")  
print(math.sqrt(error\_squared)/N)  
plt.plot(X, Y, 'o')  
#plt.plot(X, the\_g\_values)  
  
print(the\_g\_values)  
oneK = [-1.86353116e+00, -1.72499007e+00, -1.55933863e+00, -1.41138876e+00,  
 -1.25491210e+00, -1.10326055e+00, -9.32783002e-01, -7.95049052e-01,  
 -6.26173297e-01, -4.88548526e-01, -3.24043938e-01, -1.77784250e-01,  
 -9.79169909e-03, 1.20823327e-01, 3.00425833e-01, 4.36228979e-01,  
 6.03208553e-01, 7.35729567e-01, 9.08439812e-01, 1.05268401e+00,  
 1.22390520e+00, 1.36076941e+00, 1.52818496e+00, 1.66480363e+00,  
 1.83492827e+00, 1.98356599e+00, 2.14901254e+00, 2.29301155e+00,  
 2.46194535e+00, 2.59363400e+00, 2.76184567e+00, 3.28416608e+00,  
 3.45605028e+00, 3.96737719e+00, 4.52025733e+00, 4.65027953e+00,  
 5.19936236e+00, 5.34524127e+00, 5.88483224e+00, 6.40500955e+00,  
 6.57985184e+00, 7.08814764e+00, 7.25718647e+00, 7.78572438e+00,  
 8.32503742e+00, 8.46505131e+00, 9.01032470e+00, 9.14960334e+00,  
 9.70328990e+00, 1.02168126e+01]  
twoK = [ 1.28066964, 1.16695908, 1.03915701, 0.93252524, 0.82746184, 0.73320508,  
 0.63614008, 0.56459266, 0.4852531, 0.42742631, 0.36635367, 0.31941479,  
 0.27405009, 0.24509469, 0.21430184, 0.19795456, 0.1860416 , 0.18301425,  
 0.18760547, 0.19884343, 0.22092995, 0.24541274, 0.28360982, 0.32150662,  
 0.37714817, 0.43343462, 0.50450108, 0.57357069, 0.66316102, 0.73940997,  
 0.84497517, 1.23117613, 1.37758891, 1.86971359, 2.49711003, 2.659035,  
 3.40322607, 3.61735781, 4.46930434, 5.3798685, 5.70560598, 6.70880892,  
 7.0609749, 8.22180573, 9.49956125, 9.84668854, 11.25905916, 11.63525902,  
 13.19293126, 14.72635761]  
threeK = [ 0.75524347, 0.72135195, 0.68233573, 0.64904259, 0.61560302, 0.58510171,  
0.55326309, 0.52958731, 0.50325129, 0.48413079, 0.46422038, 0.44937218,  
0.43583238, 0.428039 , 0.42143878, 0.41976356, 0.42181004, 0.42679438,  
0.43795017, 0.45147107, 0.47269578, 0.4938523 , 0.52498587, 0.55482148,  
0.59773462, 0.64062838, 0.69449487, 0.74678959, 0.81475931, 0.87284005,  
0.9537104 , 1.25493832, 1.37145884, 1.77234037, 2.30292498, 2.44319964,  
3.10428595, 3.29926327, 4.09464074, 4.97721197, 5.30057741, 6.32029655,  
6.68644653, 7.92186099, 9.32955657, 9.7202182 , 11.34424854, 11.78588089,  
13.6531791 , 15.54971264]  
fourK = [ 0.79286341, 0.74946145, 0.70048067, 0.65950213 , 0.61909671, 0.58290676,  
 0.54582862, 0.51874553, 0.48915674, 0.46807837, 0.44657105, 0.4309213,  
 0.41710528, 0.40952449, 0.40375508, 0.40307244, 0.40670864, 0.41320332,  
 0.42660098, 0.44218669, 0.46603517, 0.48938642, 0.52327738, 0.55539039,  
 0.60113292, 0.64645955, 0.70295502, 0.75744307, 0.82784261, 0.88769051,  
 0.97063254, 1.27671748, 1.39424624, 1.79621717, 2.32466746, 2.46395126,  
 3.11898448, 3.31189896, 4.09838561, 4.97128676, 5.29137111, 6.30217754,  
 6.66575958, 7.89538983, 9.30245653, 9.69413786, 11.32816713, 11.77414706,  
 13.66760301, 15.60357669]  
fiveK = [ 1.74031323e+00, 1.34649198e+00, 9.57010618e-01, 6.76469700e-01,  
 4.41231674e-01, 2.66746616e-01, 1.25661102e-01, 4.88409313e-02,  
 -6.33226558e-03, -2.39363096e-02, -1.78317964e-02, 8.36588151e-03,  
 5.79996104e-02 , 1.08177995e-01, 1.89797733e-01, 2.58699796e-01,  
 3.49060404e-01 , 4.23441126e-01, 5.21729231e-01, 6.03454370e-01,  
 6.98405922e-01 ,7.71755156e-01, 8.57481054e-01, 9.23684149e-01,  
 1.00099298e+00 , 1.06372798e+00, 1.12833511e+00, 1.18030669e+00,  
 1.23669351e+00, 1.27766780e+00, 1.32696887e+00, 1.47056528e+00,  
 1.51969914e+00, 1.69516051e+00, 1.97927620e+00, 2.06699686e+00,  
 2.55256930e+00, 2.71735134e+00, 3.47753925e+00, 4.45338218e+00,  
 4.83665279e+00, 6.10311170e+00, 6.57089147e+00, 8.15536685e+00,  
 9.89009592e+00, 1.03444154e+01, 1.20459092e+01, 1.24457490e+01,  
 1.37689390e+01, 1.44000749e+01]  
sevenK = [ 1.81507344, 1.42854078, 1.02555, 0.72021757, 0.45230011, 0.24467905,  
 0.06910266, -0.03109022, -0.10751449, -0.13560756, -0.13367936, -0.10376381,  
 -0.04197174, 0.0227304 , 0.13012642, 0.22182477, 0.3425589 , 0.44177011,  
 0.57182023, 0.67834263, 0.79927975, 0.8898241 , 0.99133496, 1.06559637,  
 1.14637174, 1.20593338, 1.26016564, 1.29739721, 1.33013657, 1.34835097,  
 1.36382168, 1.38064104, 1.38584229, 1.44626875, 1.6678111 , 1.75426126,  
 2.29932421, 2.4973017 , 3.43154584, 4.61369368, 5.06314884, 6.47363204,  
 6.96403874, 8.50263073, 9.98818954, 10.34917357, 11.65654399, 11.97385145,  
 13.2972798 , 14.98075622]  
nineK = [ 2.04164583e+00, 1.41245749e+00, 8.79454291e-01, 5.52797651e-01,  
 3.17757176e-01, 1.67829361e-01, 6.34641233e-02, 1.43386337e-02,  
 -1.53946501e-02, -2.09778828e-02, -1.05496188e-02, 1.18818491e-02,  
 5.12379663e-02, 9.12542305e-02, 1.59117774e-01, 2.19863598e-01,  
 3.04917922e-01, 3.79775774e-01, 4.85594768e-01, 5.79568862e-01,  
 6.95360906e-01, 7.89318922e-01, 9.03371812e-01, 9.93761358e-01,  
 1.10038453e+00, 1.18610362e+00, 1.27130389e+00, 1.33539238e+00,  
 1.39754489e+00, 1.43588059e+00, 1.47207456e+00, 1.50688733e+00,  
 1.50156679e+00, 1.49230100e+00, 1.61355785e+00, 1.67952964e+00,  
 2.17688221e+00, 2.37552323e+00, 3.36128915e+00, 4.63787969e+00,  
 5.11858908e+00, 6.58533378e+00, 7.07693829e+00, 8.55496957e+00,  
 9.92405039e+00, 1.02608294e+01, 1.15749543e+01, 1.19284704e+01,  
 1.34731705e+01, 1.49259018e+01]  
twentyK = [ 1.95974251, 1.42955688, 0.96074577, 0.62330481 , 0.3461459 , 0.15791405,  
 0.029838 , -0.02464725 ,-0.05258808, -0.05574393, -0.04443096, -0.02201601,  
 0.0203721 , 0.06782224 , 0.15523566, 0.23700354, 0.35111612, 0.44724608,  
 0.57192738, 0.66976443 , 0.77359549, 0.84583897, 0.92265908, 0.9786285,  
 1.04448179, 1.10223263 , 1.16953943, 1.23117171, 1.30535853, 1.36218845,  
 1.42955492, 1.56814801 , 1.59178725, 1.66414068, 1.81432106, 1.85354496,  
 2.03544768, 2.12128044 , 2.92088595, 4.64296788, 5.3269541 , 6.95672412,  
 7.30401935, 8.23567223 , 9.76909479, 10.13162394, 11.48252829, 12.36555112,  
 13.30743338, 14.94947192]  
plt.plot(X, oneK)  
plt.plot(X, twoK)  
plt.plot(X, threeK)  
plt.plot(X, fourK)  
plt.plot(X, fiveK)  
plt.plot(X, sevenK)  
plt.plot(X, nineK)  
plt.plot(X, twentyK)

***For K = 1, 2, 3, 4, 5, 7, 9, 20, 30***

***Chart, line chart

Description automatically generated***

***Last Bullet***

To create random linear systems I modified code from a previous homework assignment and tracked the times for N = 20\*i for i = 1, 2, … , 50 with a for loop. Once I had my computation times I plotted them against the values on N.

import time  
import math  
import matplotlib.pyplot as plt  
import numpy as np

for i in range(1,50+1):  
 N = 20\*i  
 A = np.random.random((N,N))  
 b = np.random.random((N,1))  
 tic = time.time()  
 x = np.linalg.solve(A, b)  
 toc = time.time()  
 print(toc - tic)

Once I had my computation times, I plotted them against the values on N.

Chart, scatter chart

Description automatically generated

As you can see there are some outliers.

Next, I used my polynomial curve fit to fit the curve.

Chart, line chart

Description automatically generated

The outliers mess with the range of my graph but my guess is that the computation time grows very rapidly at a certain point, maybe even exponentially.

527 by 527 matrix: Well, based on the graph above, I would guess that it would take .0048 seconds

When testing the 527x527 it took about 0.01296687126159668 seconds

What size system in a week?

What size system in a year?

What size system in a 10 years?

I honestly am not sure. I don’t think that the relationship between the size of the matrix and time it takes to complete is linear.