1. Using four points on the interval [x0, x3], do the following:
2. Construct all of the Lagrange polynomials Lj(x) that correspond to the points x0, x1, x2, and x3 by hand.

L0 = (x-x1) \* (x-x2) \* (x-x3)

(x0-x1) (x0-x2) (x0-x3)

L1 = (x-x0) \* (x-x2) \* (x-x3)

(x1-x0) (x1-x2) (x1-x3)

L2 = (x-x0) \* (x-x1) \* (x-x3)

(x2-x0) (x2-x1) (x2-x3)

L3 = (x-x0) \* (x-x1) \* (x-x2)

(x3-x0) (x3-x1) (x3-x2)

1. Use the Lagrange polynomials to construct the interpolating polynomial P3(x) that interpolates the function f(x) at the points x0, x1, x2 and x3 by hand.

P3(x) = f(x0)\*L0 + f(x1)\*L1 + f(x2)\*L2 + f(x3)\*L3

1. Using the P3(x) you derived, create an interpolant for:

f(x) = sin(pi/2\*x) + x^2/4

over [x0, x3] with x0 = 0, x1 = 2, x2 = 3 and x3 = 4. You may do this using something like Python

or MATLAB but write your own functions rather than using the built-in ones. Plot the actual function and your interpolant using 100 equally spaced points for x between -0.5 and 4.5.

import numpy as np

import matplotlib.pyplot as plt

x0 = 0

x1 = 2

x2 = 3

x3 = 4

x\_points = np.array([x0, x1, x2, x3])

L0 = lambda x: (x-x1)\*(x-x2)\*(x-x3) / ((x0-x1)\*(x0-x2)\*(x0-x3))

L1 = lambda x: (x-x0)\*(x-x2)\*(x-x3) / ((x1-x0)\*(x1-x2)\*(x1-x3))

L2 = lambda x: (x-x0)\*(x-x1)\*(x-x3) / ((x2-x0)\*(x2-x1)\*(x2-x3))

L3 = lambda x: (x-x0)\*(x-x1)\*(x-x2) / ((x3-x0)\*(x3-x1)\*(x3-x2))

f = lambda x: np.sin(np.pi/2\*x) + x\*\*2/4

P3 = lambda x: f(x0)\*L0(x) + f(x1)\*L1(x) + f(x2)\*L2(x) + f(x3)\*L3(x)

x = np.linspace(-0.5, 4.5, 100)

%matplotlib inline

plt.figure(dpi=300)

plt.plot(x, f(x), label = ‘f(x)’)

plt.plot(x, P3(x), label = ‘P3(x)’)

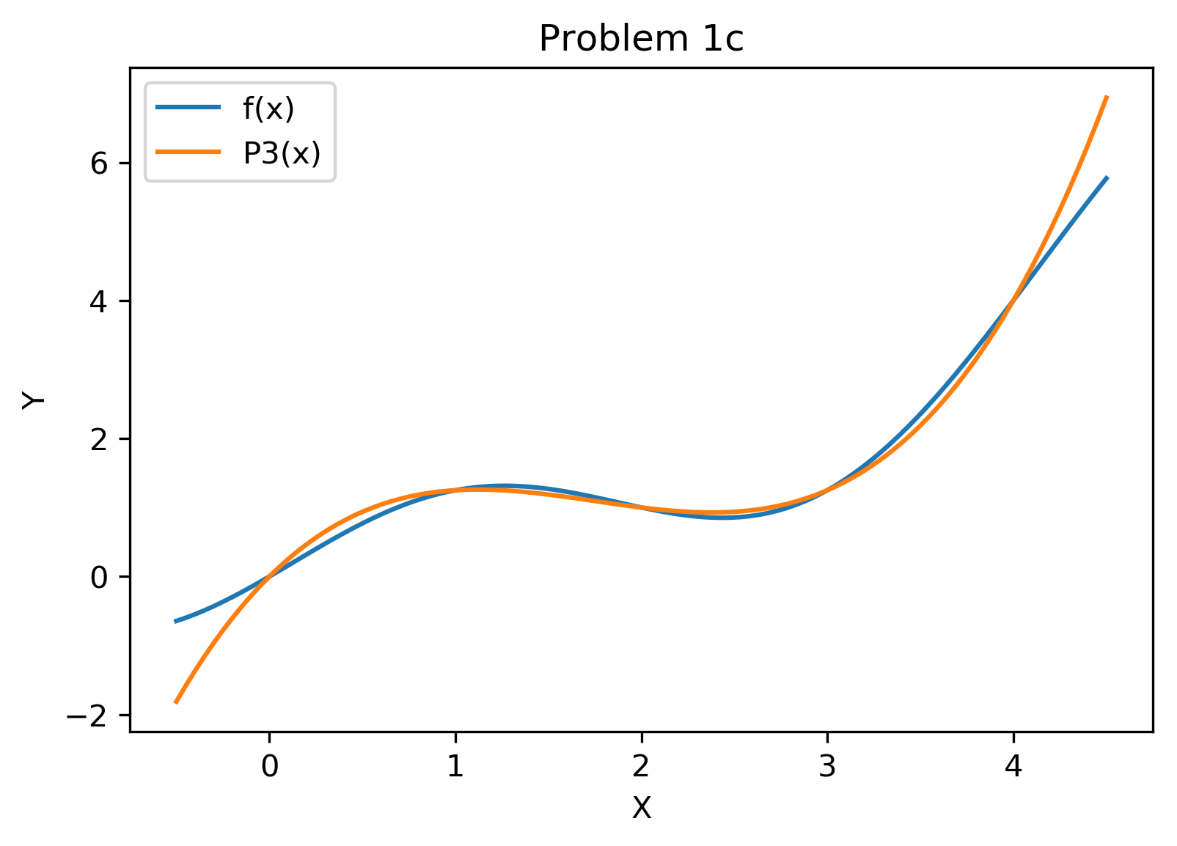
plt.xlabel("X")

plt.ylabel("Y")

plt.title("Problem 1c")

plt.legend()

plt.show()



1. Repeat what you did in part c. but instead use x0 = 0, x1 = 1, x2 = 2.5 and x3 = 4. Discuss the difference in how the function is interpolated using the different point sets.

import numpy as np

import matplotlib.pyplot as plt

x0 = 0

x1 = 1

x2 = 2.5

x3 = 4

x\_points = np.array([x0, x1, x2, x3])

L0 = lambda x: (x-x1)\*(x-x2)\*(x-x3) / ((x0-x1)\*(x0-x2)\*(x0-x3))

L1 = lambda x: (x-x0)\*(x-x2)\*(x-x3) / ((x1-x0)\*(x1-x2)\*(x1-x3))

L2 = lambda x: (x-x0)\*(x-x1)\*(x-x3) / ((x2-x0)\*(x2-x1)\*(x2-x3))

L3 = lambda x: (x-x0)\*(x-x1)\*(x-x2) / ((x3-x0)\*(x3-x1)\*(x3-x2))

f = lambda x: np.sin(np.pi/2\*x) + x\*\*2/4

P3 = lambda x: f(x0)\*L0(x) + f(x1)\*L1(x) + f(x2)\*L2(x) + f(x3)\*L3(x)

x = np.linspace(-0.5, 4.5, 100)

%matplotlib inline

plt.figure(dpi=300)

plt.plot(x, f(x), label = 'f(x)')

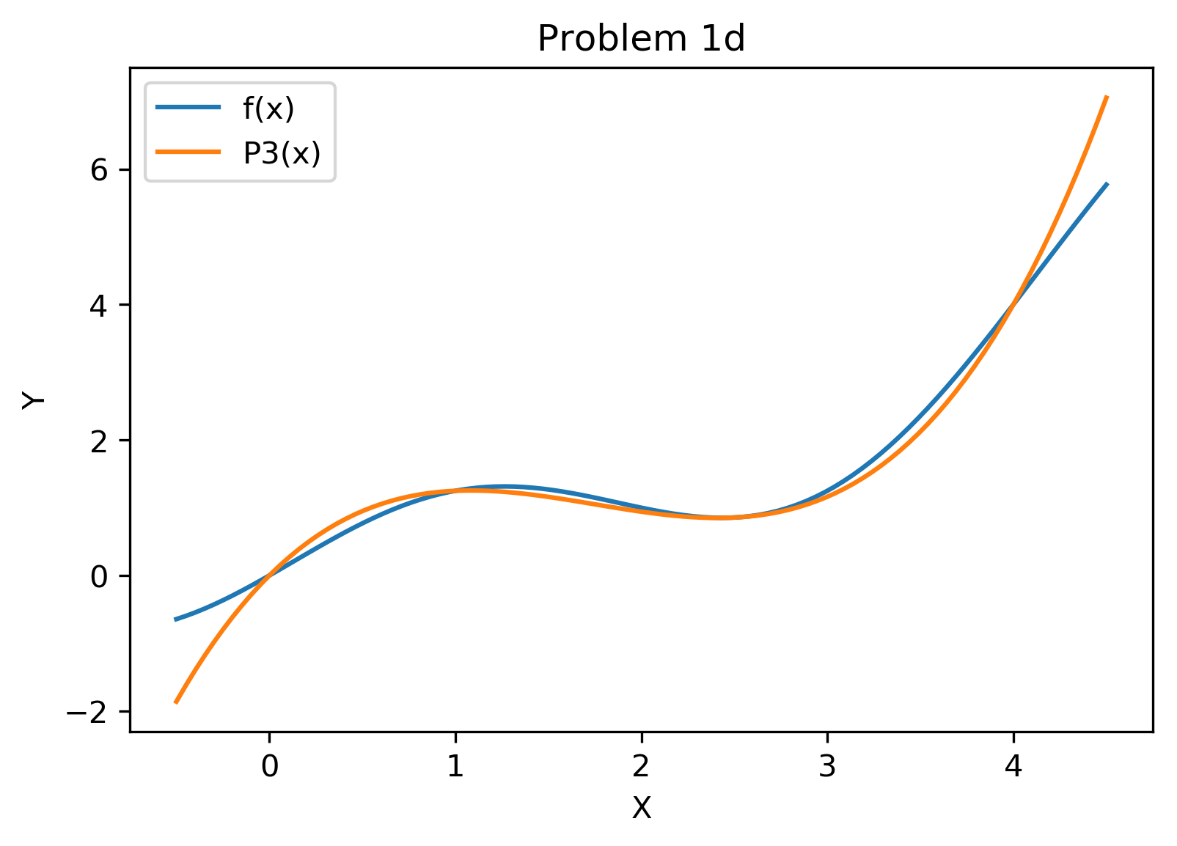
plt.plot(x, P3(x), label = 'P3(x)')

plt.xlabel("X")

plt.ylabel("Y")

plt.title("Problem 1d")

plt.legend()



You can definitely see a tighter fit for the first set of points rather than the second.

1. Using the interpolation P3(x) derived in question 1:
2. Write the general expression for the error term, err(x) = |f(x) – P3(x)|.

Four terms, so fourth derivative and four factorial:

Error = f^4(ξ) \* (x-x0)\*(x-x1)\*(x-x2)\*(x-x3)

4!

1. Given f(x) = sin(π/2\*x) +x^2/4 use information about the function to bound the error expression.

Given the function, you will take the fourth derivative of the function to help bound the expression. The fourth derivative of sin(π/2\*x) + x\*\*2/4 is:

π^4\*sin(π/2\*x)

16\*4!

1. Use the values x0 = 0, x1 = 2, x2 = 3 and x3 = 4 to get the upper bound of err(x) over this interval. That is, insert the points into the expression from part b, find an expression for x that maximizes error, and then find the x that gives the maximum. Present one final number. You may use a mathematical package to assist you in solving for x.

Plug in values given:

Errormax = (π^4/16\*sin(π/2\*€))/16\*4! \* (x) \* (x-2) \* (x-3) \* (x-4)

Maximum value for sin at the interval of [0,4] is 1 when sin = pi/2, thus x must equal 1. For the polynomial portion of the equation, the maximum value of x=2.4691. This gives the equation he following values:

π^4\*sinπ/2)\*2.4691\*(2.4691-2)\*(2.4691-3)\*(2.4691-4), which equals 0.2388

16\*4\*3\*2\*1

1. We have the following data: x = [1,2,3,4,5,6,7] and f(x) = [1,4,10,12,5,4,0]
2. Using built-in Python or MATLAB functions, interpolate this data using
   * Piecewise linear interpolation
   * Lagrange polynomial interpretation
   * Spline interpretation

Create a subplot for each of your interpolants over [0.75, 7.25] using a fine mesh spacing, e.g. 0.05 (note that to use scipy’s piecewise linear polynomial interpretation you will need to restrict the range to the exact endpoints, 1.0 and 7.0). Include the data points on the interpolation plots.

import matplotlib.pyplot as plt

from scipy import interpolate

x = [1,2,3,4,5,6,7]

y = [1,4,10,12,5,4,0]

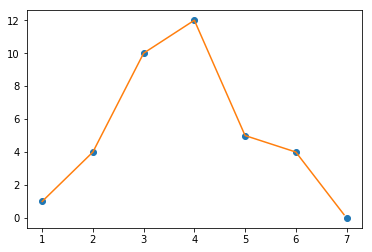
f = interpolate.interp1d(x, y)

xnew = np.arange(1,7,.05)

ynew = f(xnew)

plt.plot(x, y, 'o', xnew, ynew, '-')

plt.show()



import numpy as np

import matplotlib.pyplot as plt

from scipy.interpolate import lagrange

x = (1,2,3,4,5,6,7)

y = (1,4,10,12,5,4,0)

poly = lagrange(x, y)

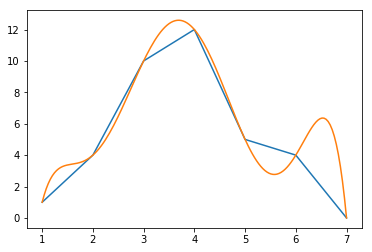
from numpy.polynomial.polynomial import Polynomial

xs=np.linspace(1.0,7.0,150)

f=poly(xs)

plt.plot(x,y)

plt.plot(xs,f)



import numpy as np

x = (1,2,3,4,5,6,7)

y = (1,4,10,12,5,4,0)

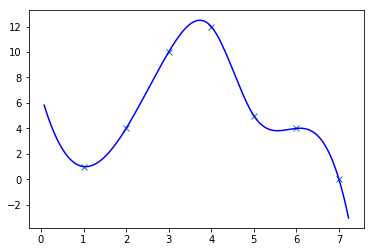
tck = interpolate.splrep(x, y, s=0)

xnew1 = np.arange(.075,7.25,0.05)

ynew1 = interpolate.splev(xnew1,tck,der=0)

plt.figure()

plt.plot(x, y, 'x', xnew1, ynew1, 'b')



1. Briefly discuss the differences between the resulting interpolations.

The piecemeal interpretation plotted straight lines between each point, whereas the spline actually calculated a function to approximate the curve between all of the points. The Lagrange varied wildly – you can see the spike around x=4 nearing -150 and x=5 nearing 150. This is quite the bizarre solution.

1. The errors generated by a numerical method on a test problem with various grid resolutions have been recorded in the following table:

|  |  |
| --- | --- |
| Grid Spacing (h) | Error (E) |
| 5.00000e-02 | 1.036126e-01 |
| 2.50000e-02 | 3.333834e-02 |
| 1.25000e-02 | 1.375409e-02 |
| 6.25000e-03 | 4.177237e-03 |
| 3.12500e-03 | 1.103962e-03 |
| 1.56250e-03 | 2.824698e-04 |
| 7.81250e-04 | 7.185644e-05 |
| 3.90625e-05 | 1.813937e-05 |

For this numerical method, the error should be in the form E = k\*h^p

1. Write this problem as a linear system Ax=b where x = (ln(k)/p) is the vector of unknowns.

E = k\*h^p h is the power variable, k is the scalar, p is the power

Take ln of both sides

ln(E) = ln(k) \* ln(h^p) ln (h^p) = p\*ln(h)

Linear system equals ln(E) = ln(k) + p\*ln(h)

Divide both sides by p: ln(E) = ln(k)

1. Derive the normal equations for this over-determined system: write the matrices in Ax=b form, where you include formulas / values for each entry.

E^2 = error. Error equals the sum of the left-hand side minus the right-hand side for all values, squared. Thus:

8

Error = ∑ (ln(E) – (ln(k) + p\*ln(h))^2

i=1

Take partial derivatives:

8

∂e = -2 \* ∑ (ln(E) – (ln(k)+p\*ln(h)) = 0

∂a0 i=1

∂e = -2 \* ∑ (ln(E) – (ln(k)+p\*ln(h))^2 = 0

∂a1 i=1

Solve for E

∑(ln(E) = ∑ ln(k) + ∑p\*ln(h)

∑ln(E)\*ln(h) = ∑ln(h)\*ln(k) + ∑p\*ln(h)^2

Convert To matrix format:

8 ∑ln(h) ln(k) ∑ln(E)

8

∑ln(h) ∑ln(h)^2 p = ∑ln(h) \*ln(E)

i=1

1. Solve, using the program/language of your choice, the normal equations to obtain a least squares estimate to parameters k and p.

from numpy import \*

h = (5e-2,2.5e-2,1.25e-2,6.25e-3,3.125e-3,3.125e-3,1.5625e-3,7.8125e-4,3.90625e-4)

E = (1.036126e-1,3.333834e-2,1.375409e-2,4.177237e-3,1.103962e-3,1.103962e-3,2.824698e-4,7.185644e-5,1.813937e-5)

c = log(h)

a = sum(c)

m=8

A=matrix([[m,a],[a,a\*\*2]])

e=sum(log(E))

B=matrix([[e],[e\*a]])

linalg.solve(A,B)

Out: matrix([[-2.43372535e-15],

[ 1.17374588e+00]])

1. Solve for the parameters k and p using scipy’s CurveFit function.

from numpy import \*

from math import \*

h = (5e-2,2.5e-2,1.25e-2,6.25e-3,3.125e-3,3.125e-3,1.5625e-3,7.8125e-4,3.90625e-4)

E = (1.036126e-1,3.333834e-2,1.375409e-2,4.177237e-3,1.103962e-3,1.103962e-3,2.824698e-4,7.185644e-5,1.813937e-5)

from scipy.optimize import \*

def f(h, k, p):

return (k\*h\*\*p)

params = curve\_fit(f,h,E) #Error in here somewhere

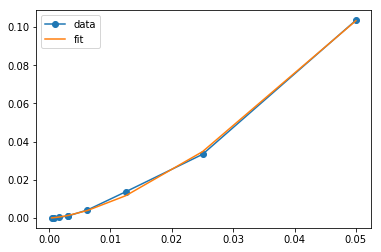
print("k=%g, p=%g" %(params[0][0],params[0][1]))

plt.plot(x\_given, y\_given,'-o')

plt.plot(x\_given, f(x\_given,params[0][0],params[0][1]))

plt.legend(['data','fit'], loc='best')

k=11.5117, p=1.57314



1. Make a log-log and a lin-lin plot that displays both the input data and the function E = k\*h^p using both your scipy’s k and p. Comment on the differences between the two approximations. (Check out python’s matplotlib.pyplt.loglog command).

import numpy as np

import matplotlib.pyplot as plt

import matplotlib.ticker as mtick

h = (5e-2,2.5e-2,1.25e-2,6.25e-3,3.125e-3,3.125e-3,1.5625e-3,7.8125e-4,3.90625e-4)

x = np.linspace(0, 1, 1000)

y = x\*\*3

#y=k\*h^p, I need the answer for part D, to input the values for k and p

fig, ax = plt.subplots()

ax.loglog(x,y, basex=np.e, basey=np.e)

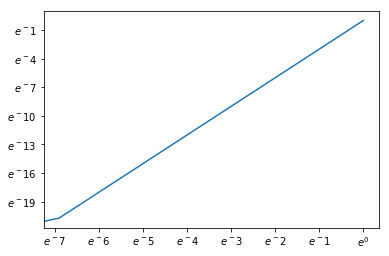
def ticks(y, pos):

return r'$e^{:.0f}$'.format(np.log(y))

ax.xaxis.set\_major\_formatter(mtick.FuncFormatter(ticks))

ax.yaxis.set\_major\_formatter(mtick.FuncFormatter(ticks))

plt.show()



BONUS: submit your code by providing read/clone access to an online version control repository where your code is stored (e.g. github or bitbucket). If you don’t know what that means and want to learn about it, come talk to me or check out resources here: <http://software-carpentry.org/lessons.html> For Windows, you want to set up the Git GUI.

COLLABORATION: I worked with CPT Drake, Maj Veigas and CPT Owens on all problems. Lansing Horan pointed me to use lambda functions for Problem 1 and helped me troubleshoot my code when it printed out a table of values instead of a graph. Lt Rao helped me with problem 3 when my Lagrange graph had crazy values from -150 to 150. Lt Rao also helped immensely with the math for problem 2. CPT Drake helped me fix my syntax for problem 4.