Edwards Ames
CSC 301 Numerical Issues

Prof. Irina Gladkova
Tuesday, October 18th 2016

Project 2

1. Suppose we wish to prepare a table of functional values of $e^x \sin(x)$ for subsequent quadratic interpolation using Newton's approach on the interval $0 \leq x \leq 2$. What base-point spacing should be used to insure that interpolation will be accurate to four decimal places for any argument in the indicated range.

This is the table for the values of x. Using Newton's approach, we choose three points.

	x_0	x_1	x_2
x	0	1	2
$f(x) = e^x \sin(x)$	$f(x_0) = 0$	$f(x_1) = e^1 \sin(1)$	$f(x_2) = e^2 \sin(2)$

Newton's divided difference formula:

$$P(x) = f[x_1] + f[x_1, x_2](x - x_1) + \ldots + f[x_1 \ldots x_n](x - x_1) \ldots (x_n - x_{n-1})$$

Now, the divided differences,

x_0	$f[x_0]$
x_1	$f[x_1]$
x_2	$f[x_2]$

$$f[x_1, x_0] = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

$$f[x_2, x_1] = \frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

$$f[x_2, x_1, x_0] = \frac{f[x_2, x_1] - f[x_1, x_0]}{x_2 - x_0}$$

Given the following tables, formulas, and points I created a python code to compute Newton's Divided Difference Formula. Then x =1 will be plugged-in to test the formula and to check whether it outputs the correct approximation.

interpolation.py

```
import math
# Quadratic Interpolation => Fuction: f(x) = e^x * sin(x)

def FindDD(xx,yy,xyValues):
    oldFx = []
    newYY = []
    i = 1
    for x,y in xyValues:
        tempy = yy[i] - y
        tempy = tempy / (xx[i] - x)
        oldFx.append(tempy)
        i+=1
        if(i >= len(xx) ):
            break;
    return updateFX(xx, yy, oldFx)
```

```python

def updateFX(xx, yy, oldFx):
    yy.remove(yy[0])
    xx.remove(xx[1])
    newFx = []
    i=0
    for i in range(len(xx)):
        newFx.append((xx[i], oldFx[i]))
        yy[i] = oldFx[i]
        i+=1
    return newFx

def computeNewtonsDDF(DDifference, x):
    X = [0.0, 1.0, 2.0]
    Y = [(0.0)]
    # Newtons Divided Difference Formula for 3 points
    # Test our formula for x = 1
    NewtonsDDF = Y[0] + DDifference[0]*(x-X[0]) + DDifference[1]*(x-X[0])*(x-X[1])
    return NewtonsDDF

def main():
    SIZE = 3
    xx = [0.0, 1.0, 2.0] #The x values chosen
    #The y values obtained by plugging x into the equation
    yy = [0.0, math.exp(1.0)*math.sin(1.0), math.exp(2.0)*math.sin(2.0)]
    # My x and y values in a list of tuples
    xyValues = [ (0.0, 0.0), (1.0, math.exp(1)*math.sin(1)),
    (2.0, math.exp(2)*math.sin(2)) ]

    # From line 45 to 54 we append the values of divided differenes into
    # a list called DDifference, which will later be use to compute Newtons D.D.F
    DDifference = []
    for j in range(SIZE - 1):
        print "xx: ", xx
        print "yy: ", yy
        xyValues = FindDD(xx,yy,xyValues)
        DDifference.append(yy[0])
        print "iteration {}:".format(j+1) , xyValues
        print " "

    x = 1 # We choose an x within the range 0 <= x <= 2
    interpolationApproximation = computeNewtonsDDF(DDifference, x)
    # Finally compute f(x) = e^x * sin(x) at x = 1
    fx_ = math.exp(x) * math.sin(x)
    # We compare both results and ensure that the interpolation is accurate to
    #   four decimal places
    print "f(x) =  ", fx_
    print "Interpolation Approximation = ", interpolationApproximation

if __name__ == '__main__':
    main()
```

Computing Newton's Divided Difference with three points and plugging x = 1 in the formula outputs the correct approximation to more than 4 correct decimal places.

```
Project 2 — -bash — 83×24
MacBook-Air-de-Enzo:Project 2 Enzo$ python interpolation.py
xx:  [0.0, 1.0, 2.0]
yy:  [0.0, 2.2873552871788423, 6.71884969742825]
iteration 1: [(0.0, 2.2873552871788423), (2.0, 4.431494410249408)]

xx:  [0.0, 2.0]
yy:  [2.2873552871788423, 4.431494410249408]
iteration 2: [(0.0, 1.0720695615352827)]

f(x) =   2.28735528718
Interpolation Approximation =  2.28735528718
MacBook-Air-de-Enzo:Project 2 Enzo$
```

2. Run the following script:

```
%  Comparison of the interpolation polynomial  Pn
%  for sin(x) in the interval [1 2] with sin(x)
%  Pn(x) = c_1 x^n + ..+ c_n x + c_{n+1}
%

  for i = 0:3
     Nx = 10*3^i;
     dx = 1/Nx;
     x = [1:dx:2]';
% Find the interpolation polynomial using
% the Vandermonde matrix
     V = vander(x);
     C = V\sin(x);  % Warning: use \, not /
% Plot the results at grid values other than the
% ones used in the interpolation.
     y = x+0.1*dx*rand(size(x));
     subplot(2,2,i+1)
     plot(y,polyval(C,y)-sin(y))
     xlabel('x')
     ylabel('Pn(x)-f(x)')
     title(['Nx = ',num2str(Nx)])
  end
 figure(gcf);
```

Do not submit the plots or the program, but answer the following questions:

a) Using the error formula derived in class, show that $|\sin(x) - Pn(x)| \to 0$ as $n \to \infty$ for $x \in [0,2]$.

b) Does this agree with your plots?

c) In practice, is larger n "good" or "bad"? Justify.

d) What might be the source of the problem with large n?

Answers:

a) We have the error formula defined as

$$\boldsymbol{E(x) = \frac{f^{n+1}(\eta)}{(n)!}\prod_{i=0}^{n}(x - x_i)}$$

If we evaluate the error at n = 1, we obtain

$$\text{E(x)} = \frac{f''(\eta)}{2}\frac{(2-0)^2}{4} = \frac{f''(\eta)}{2}$$

Now, by comparing this result to a higher number of n such as n = 10, obtaining.

$$\text{E(x)} = \frac{f^{10}(\eta)}{3628800}\frac{(2-0)^2}{4} = \frac{f^{10}(\eta)}{3628800}$$

This shows that the error approaches zero as n keeps increasing.

b) Furthermore, the four plots show something different. We can see that when n = 10 the graph between the interval $x \in [0,2]$ shows a more accurate representation of the curve. However, when n = 270 the graph becomes a straight line in the same interval.

c) A large n is useful to greatly reduce the error and a smaller n produces a better projection of the curve. This is a very complicated decision to make as it all falls down to finding the perfect balance. Having too much error is useless and having wrong representation of the curve is as well a detriment. Choosing the right value for n is a trial and error process. Comparing large and small values of n against each other and weighting the pros and cons ultimately will lead to the appropriate value for n.

d) The source of the problem with a large n is in the interval. For $x \in [0,2]$ is a very small space for a large value of n. Each point being close to each other negatively affects the graph.

3.

In studies of radiation-induced polymerization, a source of gamma rays was employed to give measured doses of radiation. However, the dosage varied with position in the apparatus, with these figures being recorded;

Position, in. from base point	0	0.5	1.0	1.5	2.0	3.0	3.5	4.0
Dosage, 10^5 rads/hr	1.90	2.39	2.71	2.98	3.20	3.20	2.98	2.74

For some reason, the reading at 2.5 inches was not reported, but the value of radiation there is needed. Fit interpolating polynomials of various degrees to the data to supply the missing information. What do you think is the best estimate for the dosage level at 2.5 inches?

For this problem I will again use the python code to calculate the quadratic interpolation using Newtown's approach and adding a few modifications. For this case we will choose the following 3 points.

	x_0	x_1	x_2
x	1.5	2.0	3.0
y	$f(x_0) = 2.98$	$f(x_1) = 3.20$	$f(x_2) = 3.20$

The reason 2.0 and 3.0 are chosen is because these two numbers surround 2.5. Now, for the third point 1.5 is chosen. There is no difference between choosing either 1.5 or 3.0 because their difference from 2.5 is the same. Mathematically choosing either one will output the same answer.

interpolation2.py   interpolation.py

```python
import matplotlib.pyplot as plt
import numpy as np
import math

def FindDD(xx,yy,xyValues):
    oldFx = []
    newYY = []
    i = 1
    for x,y in xyValues:
        tempy = yy[i] - y
        tempy = tempy / (xx[i] - x)
        oldFx.append(tempy)
        i+=1
        if(i >= len(xx) ):
            break;
    return updateFX(xx, yy, oldFx)
```

```python
def updateFX(xx, yy, oldFx):
    yy.remove(yy[0])
    xx.remove(xx[1])
    newFx = []
    i=0
    for i in range(len(xx)):
        newFx.append((xx[i], oldFx[i]))
        yy[i] = oldFx[i]
        i+=1
    return newFx

def computeNewtonsDDF(DDifference, x):
    X = [1.5, 2.0, 3.0]
    Y = [2.98]
    NewtonsDDF = Y[0] + DDifference[0]*(x-X[0]) + DDifference[1]*(x-X[0])*(x-X[1])
    return NewtonsDDF

def createPlot():
    x = [0.0, 0.5, 1.0, 1.5, 2.0, 3.0, 3.5, 4.0]
    y = [1.90, 2.39, 2.71, 2.98, 3.20, 3.20, 2.98, 2.74]
    plt.xlabel("Position in apparatus")
    plt.ylabel("Dosage")
    plt.axis([0,4,0,4])
    plt.plot(x,y, 'rx')
    plt.grid(True)
    plt.show()

def main():
    SIZE = 3
    xx = [1.5, 2.0, 3.0]
    yy = [2.98, 3.20, 3.20]
    xyValues = [ (1.5, 2.98), (2.0, 3.20), (3.0, 3.20) ]

    DDifference = []
    for j in range(SIZE - 1):
        print "xx: ", xx
        print "yy: ", yy
        xyValues = FindDD(xx,yy,xyValues)
        DDifference.append(yy[0])
        print "iteration {}:".format(j+1) , xyValues
        print " "

    x = 2.5 #find the approximation at x = 2.5
    interpolationApproximation = computeNewtonsDDF(DDifference, x)
    print "Interpolation Approximation = ", interpolationApproximation
    # createPlot()

if __name__ == '__main__':
    main()
```

The output of the code shows that computing Newton's divided difference formula for x = 2.5, the value of 3.2733333 is obtained. This value appears to be the best estimate for the dosage level at 2.5.

```
Project 2 — -bash — 83×24
MacBook-Air-de-Enzo:Project 2 Enzo$ python interpolation2.py
xx:  [1.5, 2.0, 3.0]
yy:  [2.98, 3.2, 3.2]
iteration 1: [(1.5, 0.4400000000000004), (3.0, 0.0)]

xx:  [1.5, 3.0]
yy:  [0.4400000000000004, 0.0]
iteration 2: [(1.5, -0.2933333333333336)]

Interpolation Approximation =  3.27333333333
MacBook-Air-de-Enzo:Project 2 Enzo$
```

4.

S. H. P. Chen and S. C. Saxena report experimental data for the emittance of tungsten as a function of temperature [*Ind. Eng. Chem. Fund.* **12,** 220 (1973)]. Their data are given below. They found that the equation

$$e(T) = 0.02424\left(\frac{T}{303.16}\right)^{1.27591}$$

correlated the data for all temperatures accurately to three digits. What degree of interpolating polynomial is required to match to their correlation at points midway between the tabulated temperatures? Discuss the pros and cons of polynomial interpolation in comparison to using their correlation.

T, °K	300	400	500	600	700	800	900	1000	1100
e	0.024	0.035	0.046	0.058	0.067	0.083	0.097	0.111	0.125
T, °K	1200	1300	1400	1500	1600	1700	1800	1900	2000
e	0.140	0.155	0.170	0.186	0.202	0.219	0.235	0.252	0.269

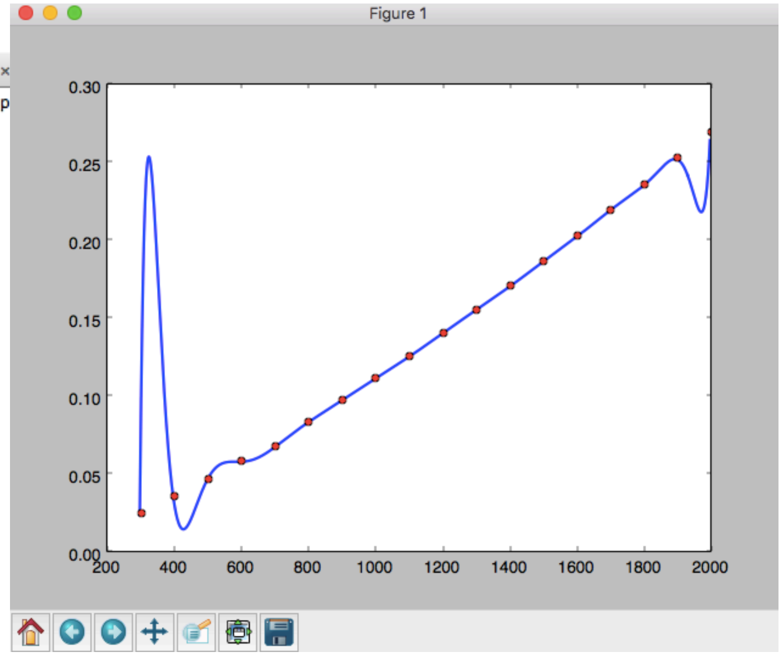
The equation given to represent their data gives a smooth curve through the points, but the accuracy on the points is only up to 3 digits (give or take). However, using polynomial interpolation the accuracy is much precise. Now, Lagrange interpolation produces a really inaccurate representation of the curve due to the large amount of data points given to the interpolation. A high degree of the resulting polynomial gives a poor prediction of the function between the points, but great accuracy at the points.

The following python code shows the interpolation with all the 18 data points

```python
import numpy as np
import matplotlib.pyplot as plt

def plot(f, points):
    x = range(300, 2000)
    y = map(f, x)
    # print y
    plt.plot( x, y, linewidth=2.0)
    xList = []
    yList = []
    for x_p, y_p in points:
        xList.append(x_p)
        yList.append(y_p)
    # print xList
    # print yList
    plt.plot(xList, yList, 'ro')

    plt.show()

def compareResult(P):
    T = 350
    while(T<2000):
        eT = 0.02424 * ((T/303.16)**1.27591)
        print "e({}) = ".format(T) + str(eT) + "\tP({}) = ".format(T) + str(P(T))
        T+=100

def lagrange(points):
    def P(x):
        total = 0
        n = len(points)
        for i in xrange(n):
            xi, yi = points[i]

            def g(i, n):
                tot_mul = 1
                for j in xrange(n):
                    if i == j:
                        continue
                    xj, yj = points[j]
                    tot_mul *= (x - xj) / float(xi - xj)
                return tot_mul

            total += yi * g(i, n)

        return total

    return P

def main():
    points = [(300,0.024), (400,0.035), (500,0.046), (600,0.058), (700,0.067),
    (800,0.083), (900,0.097), (1000,0.111), (1100,0.125), (1200,0.140),
    (1300,0.155), (1400,0.170), (1500,0.186), (1600, 0.202), (1700,0.219),
    (1800,0.235), (1900,0.252), (2000,0.269)]
    P = lagrange(points)
    compareResult(P)
    plot(P, points)
```

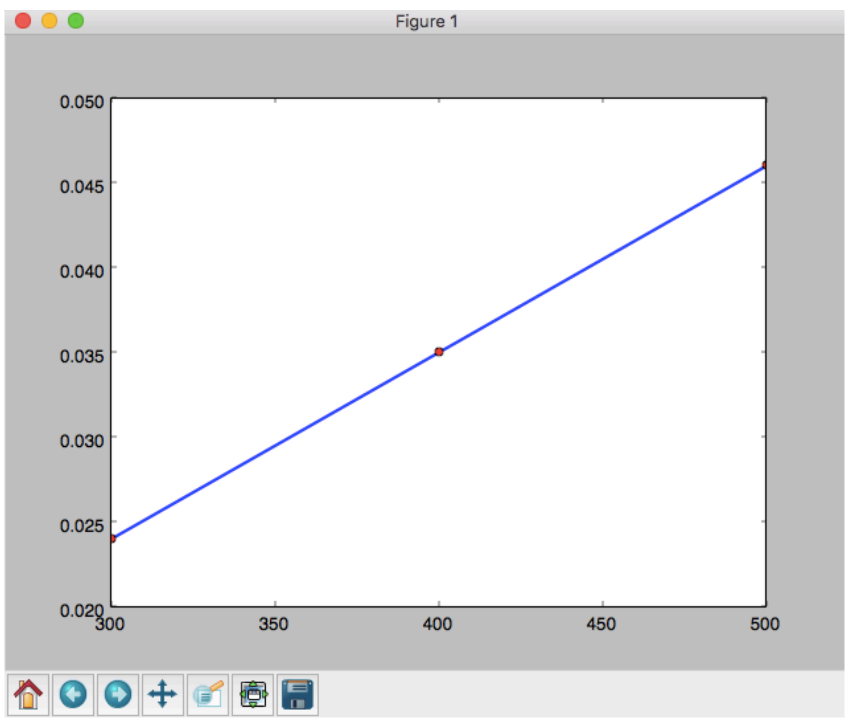
The output of the resulting polynomial

```
Project 2 — python interpolation3.py — 80×
MacBook-Air-de-Enzo:Project 2 Enzo$ python interpolation3.p
e(350) = 0.0291168562319        P(350) = 0.193473526012
e(450) = 0.0401238873158        P(450) = 0.0199512690012
e(550) = 0.0518320973674        P(550) = 0.057261757215
e(650) = 0.0641456048342        P(650) = 0.0603612549701
e(750) = 0.0769949143475        P(750) = 0.0752509348686
e(850) = 0.0903269872559        P(850) = 0.0900691062105
e(950) = 0.104099812874 P(950) = 0.104013960221
e(1050) = 0.118279168338        P(1050) = 0.117931052981
e(1150) = 0.132836555329        P(1150) = 0.132382626581
e(1250) = 0.147747818867        P(1250) = 0.147588897594
e(1350) = 0.16299218516 P(1350) = 0.162377627251
e(1450) = 0.178551569202        P(1450) = 0.177946653205
e(1550) = 0.194410062832        P(1550) = 0.193960584946
e(1650) = 0.210553547438        P(1650) = 0.210434536126
e(1750) = 0.226969395155        P(1750) = 0.226954782383
e(1850) = 0.243646234365        P(1850) = 0.245477303839
e(1950) = 0.260573762907        P(1950) = 0.229910961236
```



The output in the terminal compares both the equation given (e(x)) and the one obtained (P(x)) at points midway between the tabulated temperatures. It shows that P(x) is inaccurate at points near the head and tail of the graph.

Now, by modifying the code to compute with only 3 points; a polynomial of the 3rd degree computes an accurate representation of the midway-points between (300-500, graph below). Some other midway-points might require a 4th degree polynomial or more.




```
Project 2 — python interpolation3.py — 80×24
MacBook-Air-de-Enzo:Project 2 Enzo$ python interpolation3.py
e(350) = 0.0291168562319        P(350) = 0.0295
e(450) = 0.0401238873158        P(450) = 0.0405
```