Becs-114.1100 Computational Science – exercise round 3

Kunal Ghosh, 546247

October 11, 2015

1 Solution to Question 3 (a)

In this problem we need to compute how well a particular polynomial can interpolate a given set of data. We will also discover the effect of interpolating points on the final interpolation by the same degree polynomial.

For the first part we are required to choose 13 equally spaced points on the given polynomial $\frac{x}{\frac{1}{4}+x^2}$ The 13 chosen points have been plotted as the blue dots on 1 and a 12 degree polynomial has been interpolated as can be seen as the red line passing through all the 13 interpolating points.

However, it can be seen in 2 that the interpolated function has large errors on either end. This results in increasing the overall mean absolute error between the target function and the interpolating function, which in this case is 0.137122949967.

Note: I was just curious to figure out the reason behind higher order polynomials oscillating wildly as n (the distance from 0 on the x axis). Apparently that's because of Runge's phenomenon (`https://en.wikipedia.org/wiki/Runge's_phenomenon`) which just boils down to. I did not fully understand the lebesgue constant part. But the first point of derivatives of higher order functions increasing rapidly with increasing values of x. This essentially means, that higher order functions would be more wavy if not controlled by more interpolating points , as x grows. This is exactly what we see in this section and the next.

Any suggestions / comments that might help clear any misconceptions I might have or which might help me understand the concept better, are welcome !

The corresponding python code can be found at 3

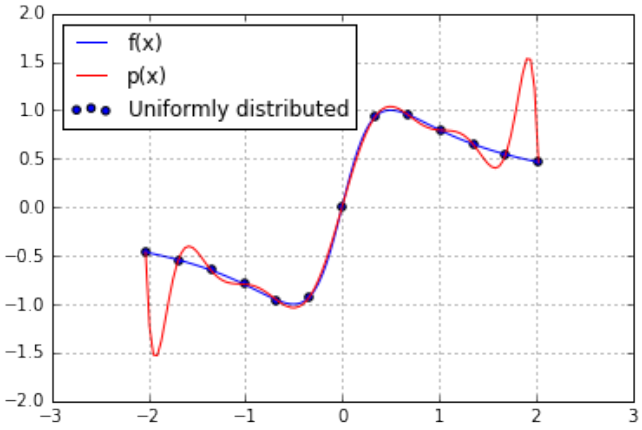

Figure 1: Plot showing the actual and the interpolation by a degree 12 polynomial over uniformly distributed points

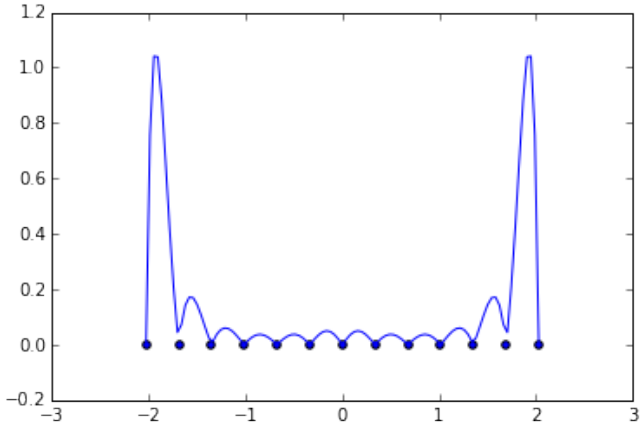

Figure 2: The absolute error $|f(x) - p(x)|$ in polynomial interpolation of $\frac{x}{(0.25+x^2)}$ using 13 equally spaced points.

2 Solution to Question 3 (b)

In this section, we use Chebychev nodes to control the fit of the polynomials at the ends. The red points in 3 are the Chebychev nodes and the blue curve is the target function. The red curve is the interpolating curve which can be visually seen to fit the target function better. This can be asserted, since the mean absolute error between the target function and the Chebychev interpolated function is 0.0272302580903 which is significantly lower that the error obtained after interpolating over uniformly distributed points. The only error we get after Interpolating using Chebychev nodes is at the central nodes 4, and that too its an order of magnitude lower than the error peaks seen in the absolute error graph when using uniformly distributed nodes in 2

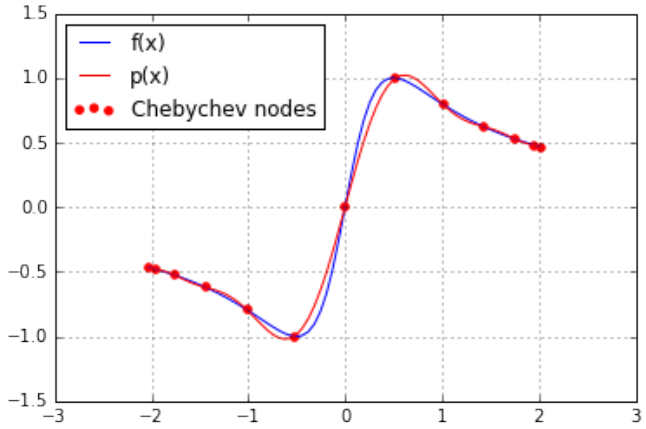

Figure 3: Plot showing the actual and the interpolation by a degree 12 polynomial over Chebychev nodes.

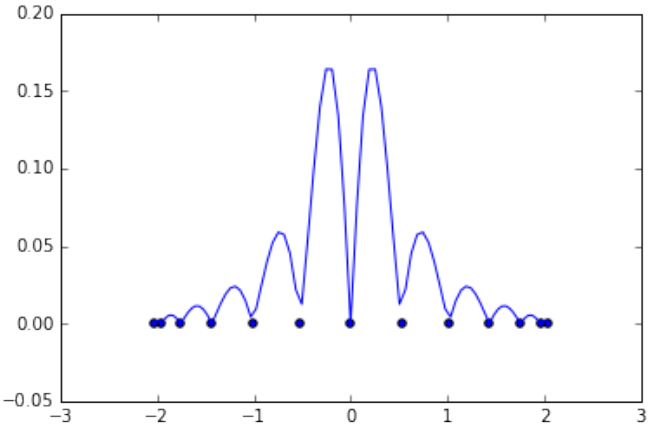

Figure 4: The absolute error $|f(x) - p(x)|$ in polynomial interpolation of $\frac{x}{0.25+x^2}$ using 13 Chebychev nodes.

The corresponding python code can be found at 4

3 Appendix A

Python source code for 1.

```
import numpy as np
import pylab as pl

# Defining the Target function
f = lambda x: x / ( 0.25 + x*x)

# Interval
xmin, xmax = -2.02857, 2.02857
n_points = 13
n = 12 # order of polynomial to fit
plot_points = 101

# 13 uniformly distributed points
points_uniform = np.linspace(xmin,xmax,n_points)
# Taking a much greater number of points to get a smooth function
points_function_plot = np.linspace(xmin, xmax, plot_points)

def coefficients(n, x, y):
    # n is the degree of the desired polynomial
    #print "x",len(x),"y",len(y)
    a = y
    for j in range(1,n+1):
        for i in range(n,j-1,-1):
            a[i] = (a[i] - a[i-1]) / (x[i] - x[i-j])
            #print "a", len(a)
    return a

def evaluate(n, x, a, t):
    pt = a[n]
    for i in range(n-1, -1, -1):
        pt = pt * (t - x[i]) + a[i];
    return pt

# Take linearly spaced X and get Y from the polynomial's analytical expression
x,y = points_uniform, f(points_uniform)

# Get Coefficients
coeff = coefficients(n, x, y)

evaluation = np.asarray([evaluate(n, x, coeff, t) for t in x], np.float)
#print evaluation

evaluation_100 = np.asarray([evaluate(n, x, coeff, t) for t in points_function_plot],
    np.float)
#print evaluation_100

# Plot the 13 points we are interested in.
pl.scatter(points_uniform, f(points_uniform),label="Uniformly distributed")
# Plot the curve.
pl.plot(points_function_plot, f(points_function_plot),label="f(x)")
pl.plot(points_function_plot, evaluation_100, color='red',label="p(x)")
# Turn on grids
pl.grid()
pl.legend(loc="upper left")
#pl.figtext(-0.05,0,"Plot showing the actual and the interpolation by a degree 12
    polynomial over uniformly distributed points")
```

```
pl.show()

# The mean absolute error
error = np.mean(np.abs(evaluation_100 - f(points_function_plot)))
print error

# 13 uniformly spaced points
pl.scatter(points_uniform, 0 * points_uniform)

#Error
error = np.abs(evaluation_100 - f(points_function_plot))
pl.plot(points_function_plot, error)
```

4 Appendix B

Python source code for 2.

```
import numpy as np
import pylab as pl

# Defining the Target function
f = lambda x: x / ( 0.25 + x*x)

# Interval
xmin, xmax = -2.02857, 2.02857
n_points = 13
n = 12 # order of polynomial to fit
plot_points = 101

# 13 uniformly distributed points
points_uniform = np.linspace(xmin,xmax,n_points)
# Taking a much greater number of points to get a smooth function
points_function_plot = np.linspace(xmin, xmax, plot_points)

def coefficients(n, x, y):
    # n is the degree of the desired polynomial
    #print "x",len(x),"y",len(y)
    a = y
    for j in range(1,n+1):
        for i in range(n,j-1,-1):
            a[i] = (a[i] - a[i-1]) / (x[i] - x[i-j])
            #print "a", len(a)
    return a

def evaluate(n, x, a, t):
    pt = a[n]
    for i in range(n-1, -1, -1):
        pt = pt * (t - x[i]) + a[i];
    return pt


def get_chebychev(n,a,b):
    '''
    A closure which returns a function which would return
    chebychev nodes for n number of points in the interval [a,b]
    '''
    def chebychev(i):
        return 0.5*(a+b) + 0.5*(b-a)*np.cos((i*np.pi)/n)
    return chebychev

# Chebychev nodes for n_points=13
chebychev = get_chebychev(n_points-1,xmin, xmax)
# range(n) => [0,n-1]
chebychev_nodes = np.asarray([chebychev(i) for i in range(n_points)], np.float)

#Chebychev nodes for plot_points=100 points
chebychev = get_chebychev(plot_points-1,xmin, xmax)
# range(n+1) => [0,n]
chebychev_nodes_100 = np.asarray([chebychev(i) for i in range(plot_points)], np.float)

#Check to ensure the 7th Chebychev node is 0
for idx, node in enumerate(chebychev_nodes):
    print idx, node
```

```
y = f(chebychev_nodes)
x = chebychev_nodes
# Get coefficients for Chebychev Nodes
coeff = coefficients(n, x, y)

cheby_eval = np.asarray(
                  [evaluate(n, chebychev_nodes, coeff, t) for t in chebychev_nodes]
                  , np.float)
cheby_eval_100 = np.asarray(
                  [evaluate(n, chebychev_nodes, coeff, t) for t in chebychev_nodes_100]
                  , np.float)

# Plot the 13 Chebychev points we are interested in.
pl.scatter(chebychev_nodes, cheby_eval, color='red', label="Chebychev nodes")

# Plot the curve.
pl.plot(chebychev_nodes_100, f(chebychev_nodes_100),label="f(x)")
pl.plot(chebychev_nodes_100, cheby_eval_100, color='red',label="p(x)")
# Turn on grids
pl.legend(loc="upper left")
pl.grid()
pl.show()

error = np.mean(np.abs(cheby_eval_100 - f(chebychev_nodes_100)))
print error

# 13 chebychev points
pl.scatter(chebychev_nodes, 0 * chebychev_nodes)

#Error
error = np.abs(cheby_eval_100 - f(chebychev_nodes_100))
pl.plot(chebychev_nodes_100, error)
pl.show()
```