Project3report

November 8, 2015

1 Project 3: Polynomial Interpolation by Arslan Memon

1.1 Part 1: Newton Interpolation

For part 1 of this project, we were required to create a method that evaluates the coefficients used for a Newton interpolation. In order to do so, I decided to look at the psuedo code in the book and use it to create the method. At first, I was having trouble with this method because I was thinking that the coefficients were a 2D matrix. I later realized that I was thinking of the augmented matrix that was used to find the coefficients. The method used nodes, x , and a matrix of the nodes evaluated using a function, y, to evaluate the coefficients using forward substitution.

Afterward, I created a method called Newton_evaluate that used the coefficients and the nodes to approximate a method at a speciified double z using Newton interpolation. I used nested multiplication to accomplish this. Afterward, I created a file called test_Newton, similar to the given test_Lagrange file, that used 11 and 21 evenly spaced nodes between 0 and 1 to test the Newton methods. These methods were used to approximate the function $cosh(2x^2)$ at 10 and 20 evenly spaced nodes between 0 and 1.

The following is the output from test_Lagrange:

```
interpolants and errors using 11 nodes:
      z        f(z)               p(z)             err
    0.050    1.0000125000260    1.0000077978837   4.7e-06
    0.150    1.0010126708709    1.0010134512525   7.8e-07
    0.250    1.0078226778257    1.0078224358909   2.4e-07
    0.350    1.0301629257234    1.0301630451041   1.2e-07
    0.450    1.0831396554576    1.0831395677641   8.8e-08
    0.550    1.1886633177627    1.1886634111673   9.3e-08
    0.650    1.3787675863905    1.3787674420731   1.4e-07
    0.750    1.7024346581382    1.7024349905050   3.3e-07
    0.850    2.2387991096881    2.2387978887376   1.2e-06
    0.950    3.1222229525487    3.1222313569473   8.4e-06

interpolants and errors using 21 nodes:
      z        f(z)               p(z)             err
    0.025    1.0000007812501    1.0000007812513   1.2e-12
    0.075    1.0000632819174    1.0000632819172   1.9e-13
    0.125    1.0004883209877    1.0004883209877   3.1e-15
    0.175    1.0018763677492    1.0018763677492   2.2e-16
    0.225    1.0051301616855    1.0051301616855   1.6e-15
    0.275    1.0114601035978    1.0114601035978   2.2e-16
    0.325    1.0223963852061    1.0223963852061   4.4e-16
    0.375    1.0398121803589    1.0398121803589   2.2e-16
    0.425    1.0659634860400    1.0659634860400   2.2e-16
    0.475    1.1035527079242    1.1035527079242         0
    0.525    1.1558250062399    1.1558250062399         0
```

```
    0.575    1.2267090054760    1.2267090054760   2.2e-16
    0.625    1.3210170862936    1.3210170862936   2.2e-16
    0.675    1.4447256306661    1.4447256306661   6.7e-16
    0.725    1.6053630078155    1.6053630078155   6.7e-16
    0.775    1.8125438090848    1.8125438090848   4.4e-16
    0.825    2.0787033700036    2.0787033700036   1.8e-15
    0.875    2.4201091598060    2.4201091598060   2.8e-14
    0.925    2.8582584261957    2.8582584261955     2e-13
    0.975    3.4218194055573    3.4218194055578   5.1e-13
```

The following is the output from test_Newton:

```
interpolants and errors using 11 nodes:
      z        f(z)               p(z)             err
    0.050    1.0000125000260    1.0000171401919   4.6e-06
    0.150    1.0010126708709    1.0010119761512   6.9e-07
    0.250    1.0078226778257    1.0078228697442   1.9e-07
    0.350    1.0301629257234    1.0301628426393   8.3e-08
    0.450    1.0831396554576    1.0831397079321   5.2e-08
    0.550    1.1886633177627    1.1886632709993   4.7e-08
    0.650    1.3787675863905    1.3787676445380   5.8e-08
    0.750    1.7024346581382    1.7024345566517     1e-07
    0.850    2.2387991096881    2.2387993638389   2.5e-07
    0.950    3.1222229525487    3.1222220146391   9.4e-07

interpolants and errors using 21 nodes:
      z        f(z)               p(z)             err
    0.025    1.0000007812501    1.0000007812507   5.6e-13
    0.075    1.0000632819174    1.0000632819174   4.4e-14
    0.125    1.0004883209877    1.0004883209877   6.2e-15
    0.175    1.0018763677492    1.0018763677492   1.3e-15
    0.225    1.0051301616855    1.0051301616855   4.4e-16
    0.275    1.0114601035978    1.0114601035978         0
    0.325    1.0223963852061    1.0223963852061         0
    0.375    1.0398121803589    1.0398121803589         0
    0.425    1.0659634860400    1.0659634860400   2.2e-16
    0.475    1.1035527079242    1.1035527079242   2.2e-16
    0.525    1.1558250062399    1.1558250062399         0
    0.575    1.2267090054760    1.2267090054760         0
    0.625    1.3210170862936    1.3210170862936         0
    0.675    1.4447256306661    1.4447256306661   2.2e-16
    0.725    1.6053630078155    1.6053630078155         0
    0.775    1.8125438090848    1.8125438090848         0
    0.825    2.0787033700036    2.0787033700036   8.9e-16
    0.875    2.4201091598060    2.4201091598060   2.2e-15
    0.925    2.8582584261957    2.8582584261957   1.5e-14
    0.975    3.4218194055573    3.4218194055575   1.7e-13
```

As can be seen by the results, both interpolation produce about the same approximations for each z value. The reason some of them are a bit different is because of round off error caused by double precision. In reality, however, both interpolations should produce the same results because interpolation between n+1 distinct nodes has a unique polynomial of degree n or less no matter what type of interpolation is used.

After testing the Newton method, we were asked to compare the speed of the two interpolations using n={10,20,40,80} and m={100,1000,10000,100000} and the function $f(x) = cosh(x^2/3)$. For each n value, I created n+1 evenly spaced nodes between -2 and 2 and tested how long it took for the coefficients to be

calculated using Newton_coeff. Then, I did 4 tests for each n using m+1 evenly spaced points between -2 and 2 and measured the time it took for Lagrange and Newton to evaluate these points and outputted the results on to the terminal. The time was measured using the chrono library.

The following are the results produced by compare.cpp:

```
Newton coefficient construction with 11 nodes took 1.5e-05 seconds.
Newton evaluation using 11 nodes and 101 z values took 8.7e-05 seconds.
Lagrange evaluation using 11 nodes and 101 z values took 0.000922 seconds.
Newton evaluation using 11 nodes and 1001 z values took 0.000578 seconds.
Lagrange evaluation using 11 nodes and 1001 z values took 0.008758 seconds.
Newton evaluation using 11 nodes and 10001 z values took 0.00525 seconds.
Lagrange evaluation using 11 nodes and 10001 z values took 0.082934 seconds.
Newton evaluation using 11 nodes and 100001 z values took 0.054631 seconds.
Lagrange evaluation using 11 nodes and 100001 z values took 0.840025 seconds.

Newton coefficient construction with 21 nodes took 3e-05 seconds.
Newton evaluation using 21 nodes and 101 z values took 0.000157 seconds.
Lagrange evaluation using 21 nodes and 101 z values took 0.00339 seconds.
Newton evaluation using 21 nodes and 1001 z values took 0.001034 seconds.
Lagrange evaluation using 21 nodes and 1001 z values took 0.033375 seconds.
Newton evaluation using 21 nodes and 10001 z values took 0.010636 seconds.
Lagrange evaluation using 21 nodes and 10001 z values took 0.312001 seconds.
Newton evaluation using 21 nodes and 100001 z values took 0.106587 seconds.
Lagrange evaluation using 21 nodes and 100001 z values took 3.10034 seconds.

Newton coefficient construction with 41 nodes took 0.000126 seconds.
Newton evaluation using 41 nodes and 101 z values took 0.000219 seconds.
Lagrange evaluation using 41 nodes and 101 z values took 0.010956 seconds.
Newton evaluation using 41 nodes and 1001 z values took 0.002672 seconds.
Lagrange evaluation using 41 nodes and 1001 z values took 0.110217 seconds.
Newton evaluation using 41 nodes and 10001 z values took 0.023518 seconds.
Lagrange evaluation using 41 nodes and 10001 z values took 1.13955 seconds.
Newton evaluation using 41 nodes and 100001 z values took 0.199885 seconds.
Lagrange evaluation using 41 nodes and 100001 z values took 11.8416 seconds.

Newton coefficient construction with 81 nodes took 0.000463 seconds.
Newton evaluation using 81 nodes and 101 z values took 0.000432 seconds.
Lagrange evaluation using 81 nodes and 101 z values took 0.048721 seconds.
Newton evaluation using 81 nodes and 1001 z values took 0.00362 seconds.
Lagrange evaluation using 81 nodes and 1001 z values took 0.475601 seconds.
Newton evaluation using 81 nodes and 10001 z values took 0.042503 seconds.
Lagrange evaluation using 81 nodes and 10001 z values took 4.62069 seconds.
Newton evaluation using 81 nodes and 100001 z values took 0.400603 seconds.
Lagrange evaluation using 81 nodes and 100001 z values took 46.337 seconds.
```

As n doubles, the time it takes to compute Newton coefficients is multiplied by approximately 4 or 5. Since Newton_coeff has a $O(n^2)$ cost, this was to be expected. The time it takes for Newton interpolations to evaluate m+1 points as n doubles is multiplied by 2 while Lagrange follows the same pattern as the Newton_coeff. As m increases, the time it takes for Newton interpolation to evaluate m+1 nodes multiplies by 10 as the number of values multiplies by 10, and Lagrange follows the same pattern. However, it takes Lagrange approximately 100 times more time to evaluate the m+1 points than it does Newton, which can be attributed to the fact that Newton evaluation has a cost of O(n), while Lagrange has a cost of $O(n^2)$ because the Lagrange basis has to be calculated for every point. Although Lagrange interpolation doesn't require coefficients to be calculated because the y values are the coefficients, it is less productive than Newton if

multiple approximations are needed for one set of nodes. In this case it is better to use Newton and calculate the coefficients once and use the evaluation function multiple times.

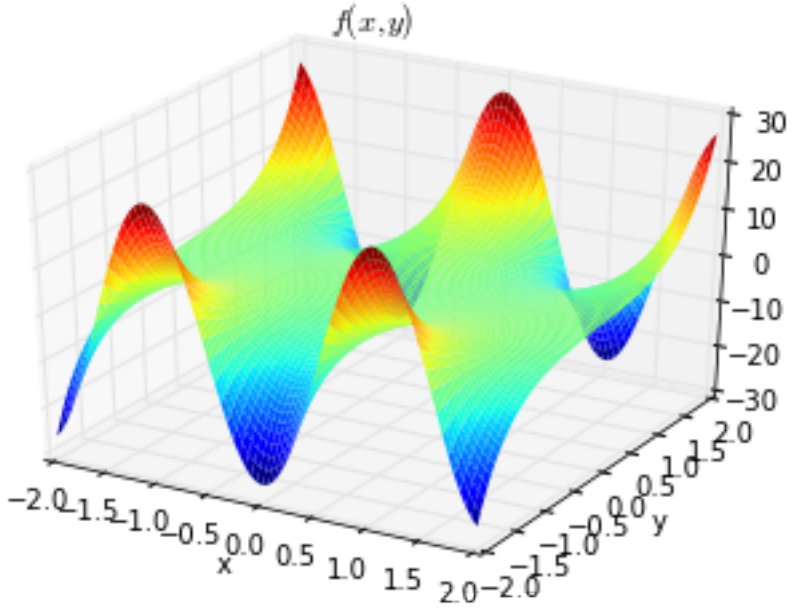
1.2 Part 2: Multi-dimensional interpolation

For part 2, I started by creating a 2D Lagrange interpolation function that took in two 1D matrices, x and y, a 2D matrix, f, and doubles, a and b. At first I had two loops that summed f(i,j) if a=x$_i$ and b=y_j, which was wrong because I was following the piecewise function in the project PDF. This function was only for the nodes used for interpolation. As a result, when I tested the function using the provided file, I ended up getting an error graph that looked more like the true function and a graph of the interpolation that was just a straight line. Also, another thing I realized was that the 2D function was running a lot faster than the 1D function, which should not be the case logically speaking.

I then used decided to use the sum of $f(i, j)$ * the Lagrange basis of a using the x nodes and b using the y nodes, which made the graphs look a lot better. If the size of x and y is the same, then the cost of 2D interpolation using Lagrange is O(n^4) because it uses two basis functions per $f(i, j)$ and loops through the 2D array. The time it took for the 2D Lagrange evaluation to finish approximating f(a,b) was fairly longer than the 1D interpolation.

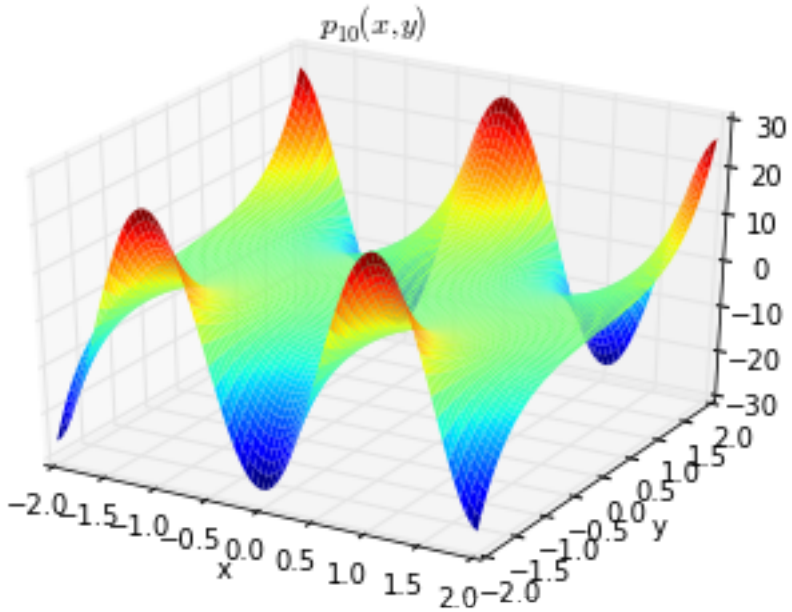
The following is the code for the graphs of test_Lagrange2D:

```
In [36]: %pylab inline
         a = loadtxt('a.txt')
         b = loadtxt('b.txt')
         f = loadtxt('ftrue.txt')
         p10 = loadtxt('p10.txt')
         p20 = loadtxt('p20.txt')

Populating the interactive namespace from numpy and matplotlib

WARNING: pylab import has clobbered these variables: ['f']
`%matplotlib` prevents importing * from pylab and numpy

In [37]: from mpl_toolkits.mplot3d.axes3d import Axes3D
         fig = figure()
         ax = fig.add_subplot(111, projection='3d')
         X, Y = meshgrid(b, a)
         surf = ax.plot_surface(X, Y, f, rstride=1, cstride=1, linewidth=0, cmap=cm.jet)
         ax.set_xlabel('x')
         ax.set_ylabel('y')
         title('$f(x,y)$')

Out[37]: <matplotlib.text.Text at 0x12c2d4c90>
```



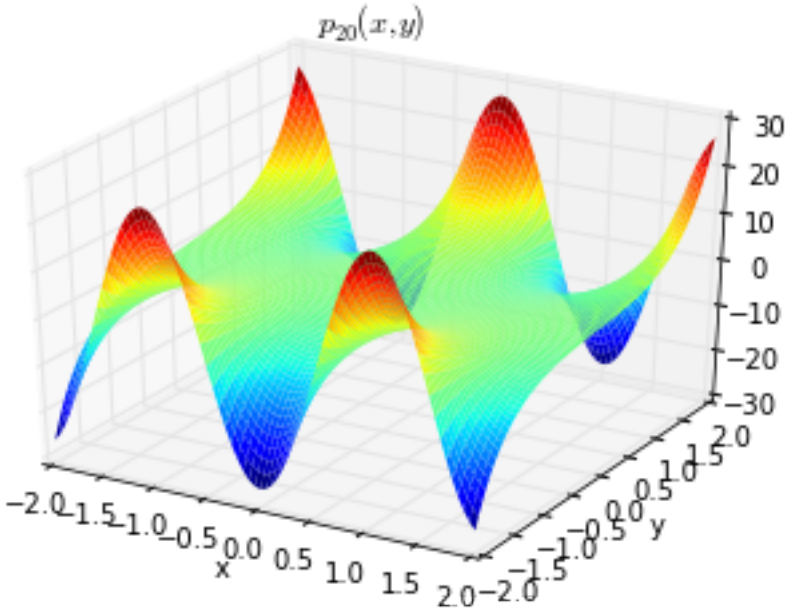
In [38]:
```
fig = figure()
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, p10, rstride=1, cstride=1, linewidth=0, cmap=cm.jet)
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$p_{10}(x,y)$')
```

Out[38]:
```
<matplotlib.text.Text at 0x12e6bd210>
```



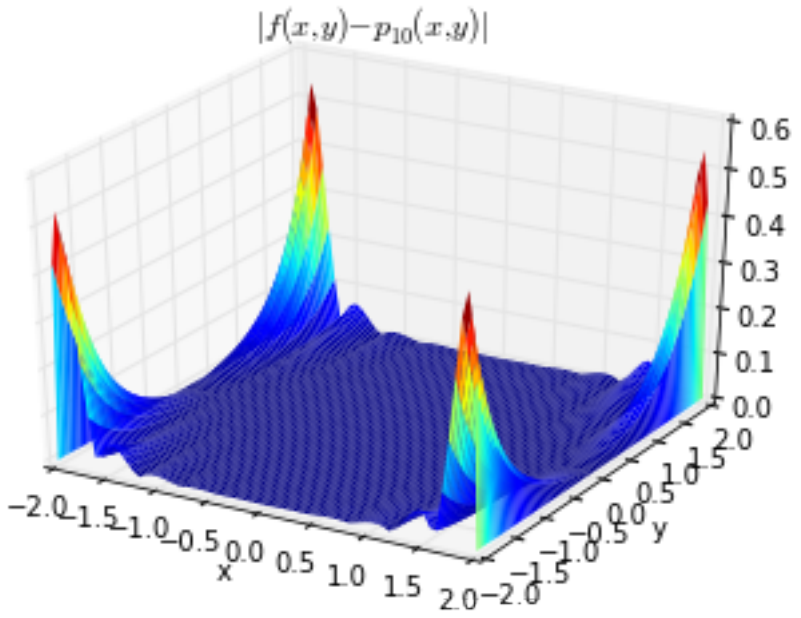
In [39]:
```
fig = figure()
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, p20, rstride=1, cstride=1, linewidth=0, cmap=cm.jet)
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$p_{20}(x,y)$')
```

Out[39]: <matplotlib.text.Text at 0x12bb8c490>



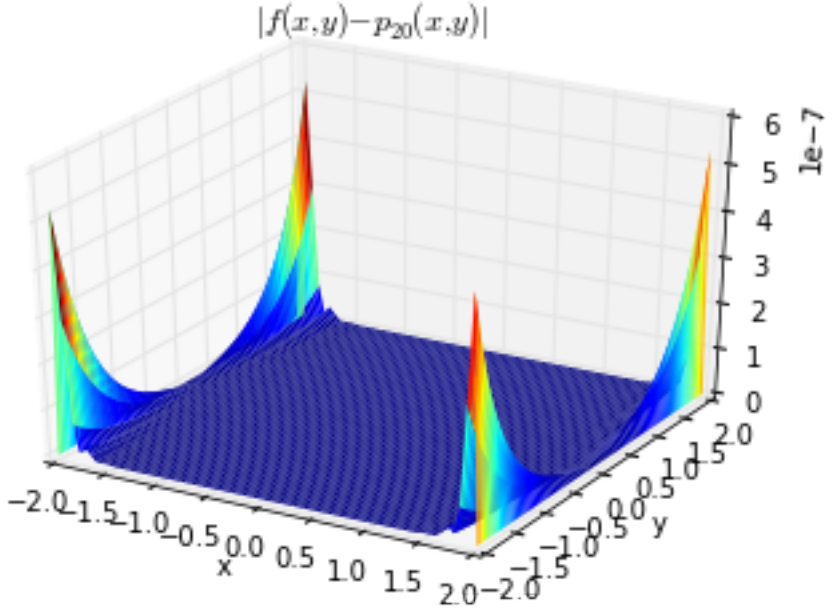
In [40]:
```
e10 = abs(f-p10)
fig = figure()
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, e10, rstride=1, cstride=1, linewidth=0, cmap=cm.jet)
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$|f(x,y) - p_{10}(x,y)|$')
```

Out[40]: <matplotlib.text.Text at 0x1317fa310>



In [41]:
```
e20 = abs(f-p20)
fig = figure()
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, e20, rstride=1, cstride=1, linewidth=0, cmap=cm.jet)
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$|f(x,y) - p_{20}(x,y)|$')
```

Out[41]: <matplotlib.text.Text at 0x1418a5550>



These error graphs seem to show that more evenly spaced nodes decrease the error, but, as will be seen in part 3, poor choices of nodes can cause major errors in a approximation.

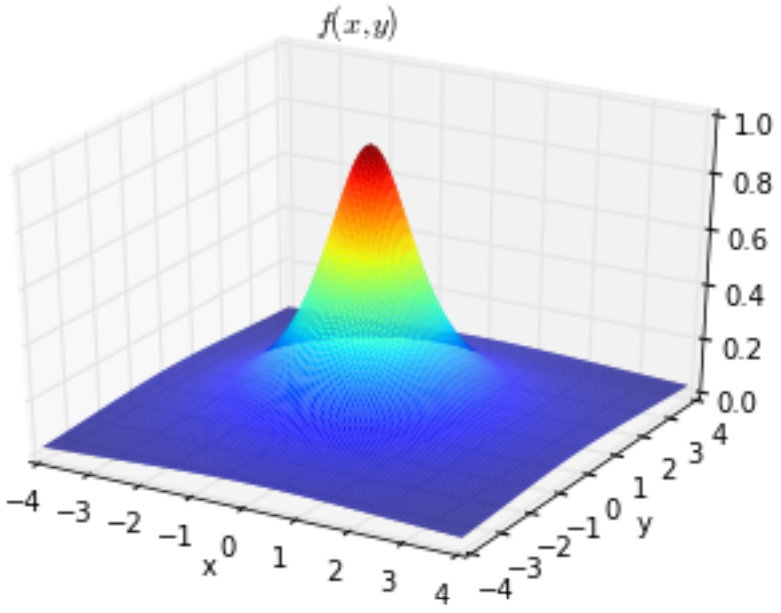
1.3 Part 3: The importance of nodes

For part 3, we were asked to approximate the 2D Runge function, $1/(1+x^2+y^2)$, using the Lagrange2D method. First, I created a file called Runge_uniform and set m=n=6 and created two matrices, x and y, with n+1 evenly spaced inputs between [-4,4]. These were the nodes of the interpolation. Then I created a m x n matrix, f, that had inputs calculated by the Runge function. Then, I created two matrices, a, which had 201 evenly spaces points between [-4,4], and b, which had 101 evenly spaces points between [-4,4]. I then created another matrix called p6 that was a 201 x 101 matrix and used Lagrange2D to approximate every combination of $f(a,b)$, which was a sixth degree polynomial approximation. Afterward, I stored a, b, and p6 using the write method. I repeated the same process with m=n=24.

Afterward, I created another file called Runge_Chebyshev and basically repeated the same process as above. The only difference was that I used Chebyshev nodes instead of evenly spaced nodes. The Chebyshev nodes were calculated using the formula $x_i = Lcos((2i+1)pi/(2m+2))$, where L=4. I then graphed the four graphs, two for uniform nodes and two for Chebyshev nodes, and their absolute errors.

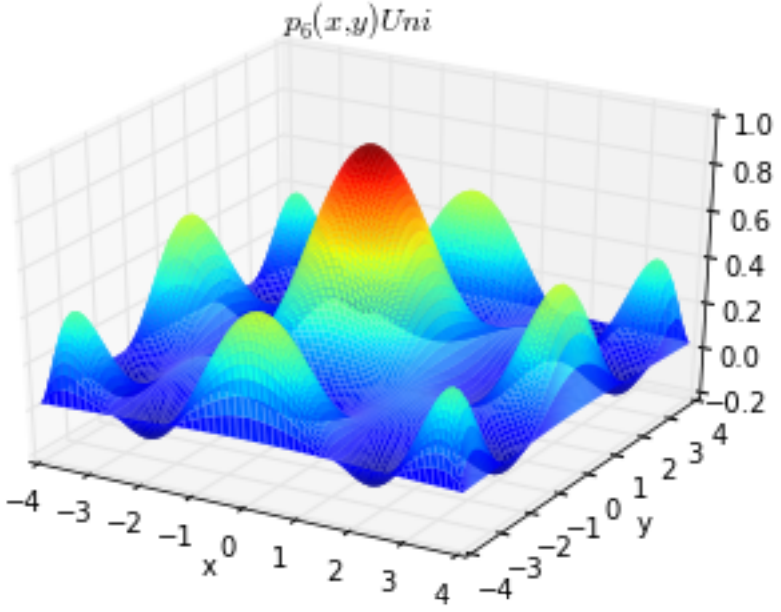
The following is the code for the graphs:

```
In [42]: %pylab inline
         a = loadtxt('avals.txt')
         b = loadtxt('bvals.txt')
         f = loadtxt('Runge.txt')
         p6_uni = loadtxt('p6_uni.txt')
         p24_uni = loadtxt('p24_uni.txt')
         p6_Cheb=loadtxt('p6_Cheb.txt')
         p24_Cheb=loadtxt('p24_Cheb.txt')

Populating the interactive namespace from numpy and matplotlib

WARNING: pylab import has clobbered these variables: ['f']
`%matplotlib` prevents importing * from pylab and numpy

In [43]: from mpl_toolkits.mplot3d.axes3d import Axes3D
         fig = figure()
         ax = fig.add_subplot(111, projection='3d')
         X, Y = meshgrid(b, a)
         surf = ax.plot_surface(X, Y, f, rstride=1, cstride=1, linewidth=0, cmap=cm.jet)
         ax.set_xlabel('x')
         ax.set_ylabel('y')
         title('$f(x,y)$')

Out[43]: <matplotlib.text.Text at 0x1330a3b90>
```



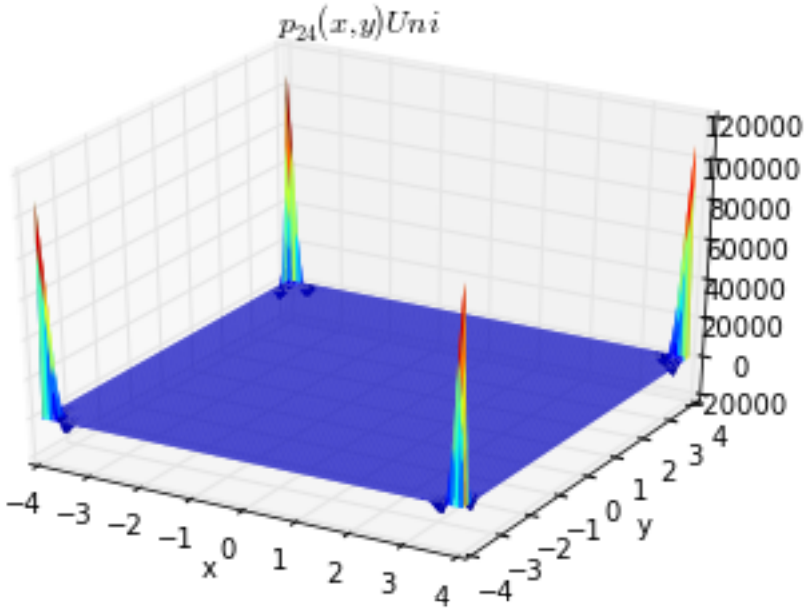
In [44]:
```
fig = figure()
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, p6_uni, rstride=1, cstride=1, linewidth=0, cmap=cm.jet)
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$p_{6}(x,y) Uni$')
```

Out[44]: <matplotlib.text.Text at 0x13cc15250>



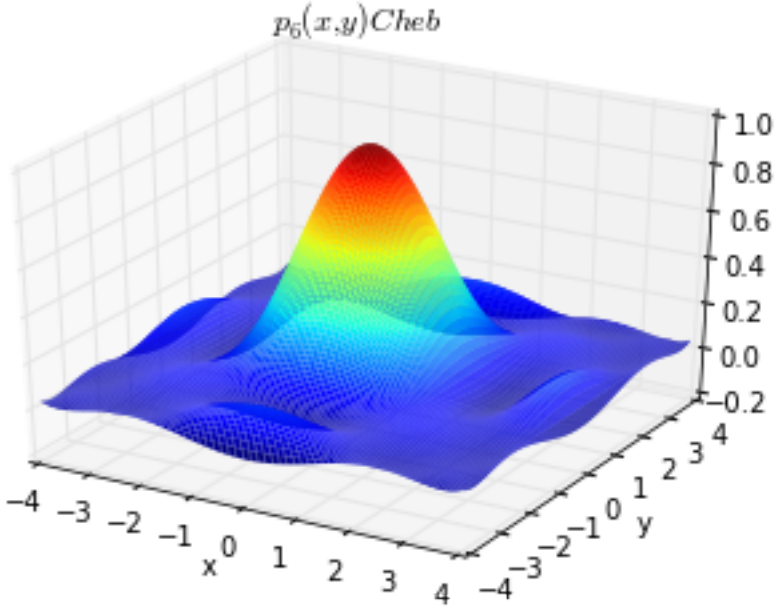
In [45]:
```
fig = figure()
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, p24_uni, rstride=1, cstride=1, linewidth=0, cmap=cm.jet)
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$p_{24}(x,y) Uni$')
```

Out[45]: <matplotlib.text.Text at 0x1418f5d10>



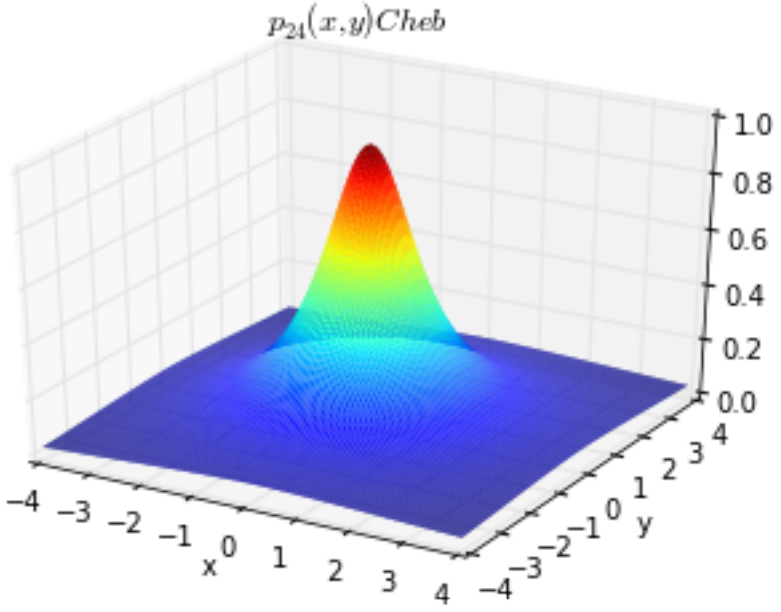
In [46]:
```
fig = figure()
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, p6_Cheb, rstride=1, cstride=1, linewidth=0, cmap=cm.jet)
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$p_{6}(x,y) Cheb$')
```

Out[46]: <matplotlib.text.Text at 0x13f0a46d0>



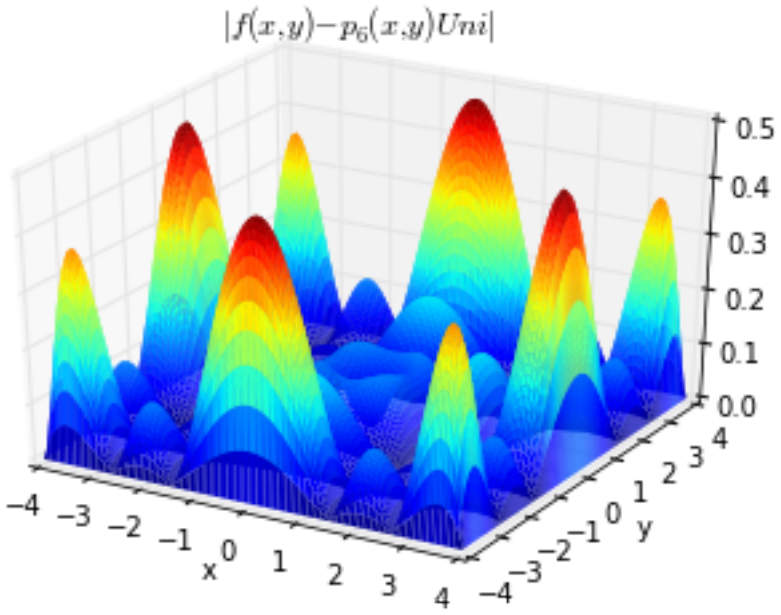
In [47]:
```
fig = figure()
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, p24_Cheb, rstride=1, cstride=1, linewidth=0, cmap=cm.jet)
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$p_{24}(x,y) Cheb$')
```

Out[47]: <matplotlib.text.Text at 0x144ae9210>



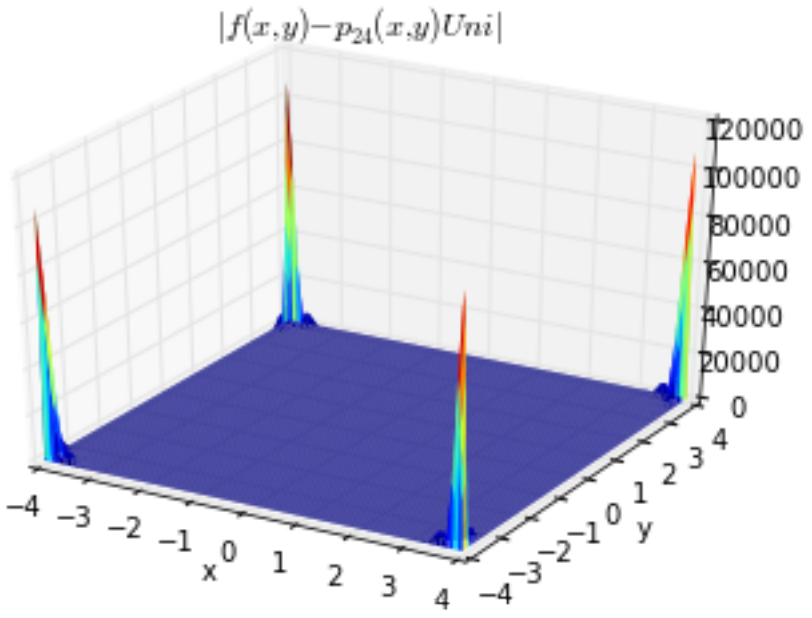
In [48]:
```
e6 = abs(f-p6_uni)
fig = figure()
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, e6, rstride=1, cstride=1, linewidth=0, cmap=cm.jet)
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$|f(x,y) - p_{6}(x,y) Uni|$')
```

Out[48]: <matplotlib.text.Text at 0x14562d590>



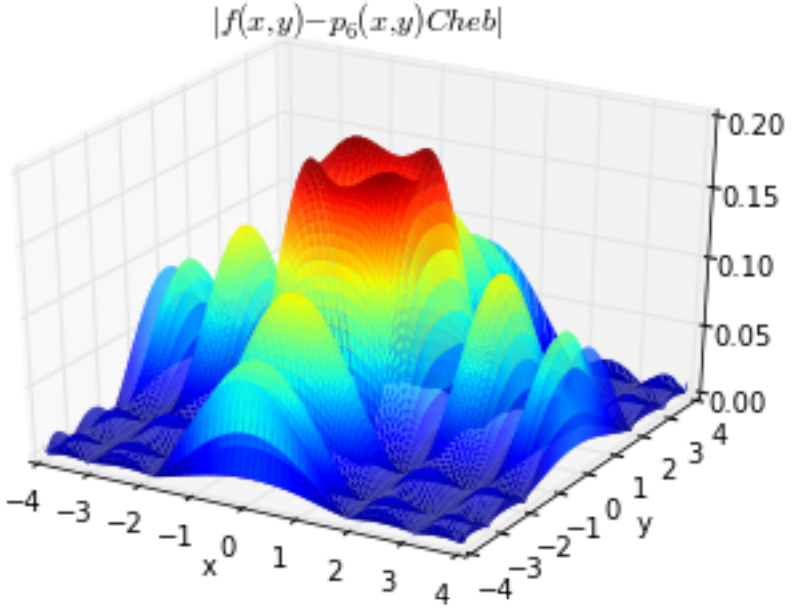
In [49]:
```
e24 = abs(f-p24_uni)
fig = figure()
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, e24, rstride=1, cstride=1, linewidth=0, cmap=cm.jet)
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$|f(x,y) - p_{24}(x,y) Uni|$')
```

Out[49]: <matplotlib.text.Text at 0x154287450>



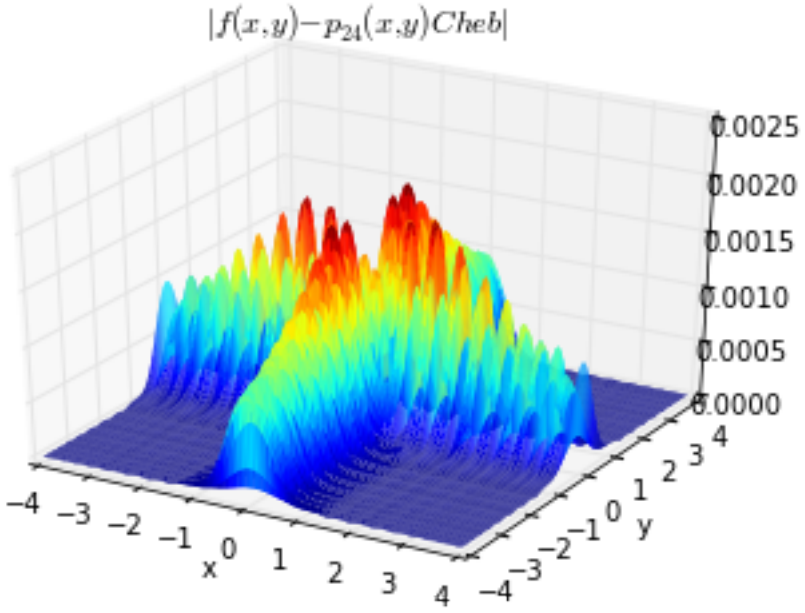
In [50]:
```
e6 = abs(f-p6_Cheb)
fig = figure()
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, e6, rstride=1, cstride=1, linewidth=0, cmap=cm.jet)
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$|f(x,y) - p_{6}(x,y) Cheb|$')
```

Out[50]:
```
<matplotlib.text.Text at 0x1445d4a10>
```



```
In [51]: e24 = abs(f-p24_Cheb)
         fig = figure()
         ax = fig.add_subplot(111, projection='3d')
         surf = ax.plot_surface(X, Y, e24, rstride=1, cstride=1, linewidth=0, cmap=cm.jet)
         ax.set_xlabel('x')
         ax.set_ylabel('y')
         title('$|f(x,y) - p_{24}(x,y) Cheb|$')
```

```
Out[51]: <matplotlib.text.Text at 0x147e372d0>
```



The error plots show that the more evenly spaced used to approximate the Runge function, the bigger the error, however, the opposite seems to be true for the Chebyshev nodes. To understand this, we can use the 1D error function bound to calculate the interpolation error bound of $1/(1+x^2)$. The first part of the error bound is $f^{n+1}(x)/(n+1)!$. In this case $|f^{n+1}(x)| = (2x)^{n+1}/(1+x^2)^{n+2})$, which will have a bound that's less than 1, which mean dividing by (n+1)! will decrease as n increases. The second part of the error calculation is the product of the evaluation point minus each node. For evenly spaced nodes, this term becomes a polynomial of degree n. The closer the evaluation point is to the endpoints, the oscillations caused by the error polynomial increases, causing the error to become huge for high degree interpolations. This can be seen in the error graph of p26 uniform, which has an error as high as 100000.

The same error bound can be used to describe the error for Chebyshev nodes. The only difference is the product of the evaluation point minus the nodes. Chebyshev nodes are denser towards the endpoints, thus the error is spread evenly throughout the nodes because they are found using the unit circle. As a result, the error decreases as the number of nodes increase. The graph of this error term has even oscillations throughout. As a result, the quality of these approximations from best to worst is p24 Cheb, p6 Cheb, p6 Uni, and p24 uni.

1.4 Code

1.4.1 Newton_interpolant.cpp

//

```
//  Newton_interpolant.cpp
//  project3
//
//  Created by Arslan Memon on 10/15/15.
//  Copyright (c) 2015 Arslan. All rights reserved.
//

#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <math.h>
#include "matrix.hpp"

using namespace std;
Matrix Newton_coefficients(Matrix& x, Matrix& y)// find coefficients for interpolation
{
    if(x.Size()!=y.Size())//not possible to find coefficients if
    //there isn't a y for every x
    {
        return Matrix(0,0);
    }
    Matrix coeff(x.Size());//coefficiennt matrix size  is same as the number of nodes

    for (int i=0;i<x.Size();i++)//initialize coeff matrix with y values
    {
        coeff(i)=y(i);
    }
    for (int j=1;j<x.Size(); j++)// the first coefficient is the y value and
    //then forward substitution is used to find other coefficients
    {
        for(int i=x.Size()-1;i>=j;i--)
        {
            coeff(i)=(coeff(i)-coeff(i-1))/(x(i)-x(i-j));
        }
    }

    return coeff;

}
double Newton_evaluate(Matrix& x, Matrix& c, double z)//eval z using Newton interpolation
{
    double tmp=c(c.Size()-1);//set tmp to last coefficient in the matrix
    for (int i=c.Size()-1;i>=0;i--)//use  nested multiplication to evaluate at z
    {
        tmp=(tmp*(z-x(i)))+c(i);
    }
    return tmp;
}
```

1.4.2 test_Newton.cpp

```
//
//  test.cpp
//  project3
```

```
//
//  Created by Arslan Memon on 10/15/15.
//  Copyright (c) 2015 Arslan. All rights reserved.
//

#include <stdio.h>

#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <math.h>
#include "matrix.hpp"
using namespace std;
Matrix Newton_coefficients(Matrix& x, Matrix& y);
double Newton_evaluate(Matrix& x, Matrix& c, double z);
//same thing as test_lagrange, that's why the comments are the same.
//the only difference is that the points are evaluated using Newton interpolation
int main()
{
    auto f = [](const double x) -> double {
        return (cosh(2.0*x*x));
    };

    // array of n values for testing
    vector<size_t> nvals = {10, 20};

    // loop over tests
    for (size_t k=0; k<nvals.size(); k++) {

        // set n, output test information
        int n = nvals[k];
        cout << endl << "interpolants and errors using " << n+1 << " nodes:\n";

        // set arrays of nodes and data values
        Matrix x = Linspace(0.0, 1.0, n+1, 1);  // set column vector of nodes
        Matrix y(n+1);                          // initialize data
        for (int i=0; i<=n; i++)                // fill data
            y(i) = f(x(i));

        // set evaluation points z as midpoints between nodes
        double dx = 1.0/n;                    // set node spacing
        Matrix z = Linspace(dx/2.0, 1.0-dx/2.0, n, 1);
        Matrix c=Newton_coefficients(x, y);
        Matrix p(n);
        // evaluate the Newton polynomial at the points z, storing in p

        for (int i=0; i<n; i++)
        {
            p(i) = Newton_evaluate(x, c, z(i));
        }
        // output errors at each point
        cout << "      z        f(z)              p(z)             err\n";
        for (int i=0; i<n; i++)
            printf("   %6.3f   %16.13f   %16.13f   %7.2g\n",
```

```
                z(i), f(z(i)), p(i), fabs(f(z(i))-p(i)));
    }
    }
```

1.4.3 Compare.cpp

```
//
//  compare.cpp
//  project3
//
//  Created by Arslan Memon on 10/19/15.
//  Copyright (c) 2015 Arslan. All rights reserved.
//

#include <stdio.h>

#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <math.h>
#include "matrix.hpp"
#include <chrono>// time library
using namespace std;
double Lagrange(Matrix& x, Matrix& y, double z);
Matrix Newton_coefficients(Matrix& x, Matrix& y);
double Newton_evaluate(Matrix& x, Matrix& c, double z);

int main()
{
    //funtion cosh(x^2/3) using lamda notation

    auto f = [](const double x) -> double {
        return (cosh(x*x/3.0));

    };
    //array of n and m values for testing
    int n[4]={10,20,40,80};
    int m[4]={100,1000,10000,100000};
        for (int i=0;i<4;i++)
    {
        //initialize x and y based on n+1 evenly spaced nodes
        Matrix x=Linspace(-2,2,n[i]+1);
        Matrix y(n[i]+1);


        for (int j=0; j<n[i]+1; j++) {
            y(j)=f(x(j));
        }
        //test time it takes to evaluate coefficients of
        //n+1 nodes using Newton_coefficients
        std::chrono::time_point<std::chrono::system_clock> start, end;
        start=chrono::system_clock::now();
        start=chrono::system_clock::now();
        Matrix c=Newton_coefficients(x, y);
```

```
        end = chrono::system_clock::now();
        chrono::duration<double> elapsed_seconds=end-start;
        cout<<"Newton coefficient construction with "<<n[i]+1<<" nodes took "
        << elapsed_seconds.count() <<" seconds."<<endl;
        for(int j=0;j<4;j++)
        {
            Matrix z=Linspace(-2, 2, m[j]+1);//set of m+1 evenly spaced values
            //between -2 and 2 that are going to be
            //evaluated using Newton and Lagrange interpolation

            //measure time taken for Newton evaluation and print results

            start=chrono::system_clock::now();
            for (int k=0; k<m[j]+1; k++) {
                Newton_evaluate(x, c, z(k));
            }
            end = chrono::system_clock::now();
            elapsed_seconds = end-start;
            cout<<"Newton evaluation using "<<n[i]+1<<" nodes and "<<m[j]+1
            <<" z values took "<< elapsed_seconds.count() <<" seconds."<<endl;


            //measure time taken for Lagrange evaluation and print results
            start=chrono::system_clock::now();
            for (int k=0; k<m[j]+1; k++) {
               Lagrange(x, y, z(k));
            }
            end = chrono::system_clock::now();
            elapsed_seconds = end-start;
            cout<<"Lagrange evaluation using "<<n[i]+1<<" nodes and "
            <<m[j]+1<<" z values took "<< elapsed_seconds.count() <<" seconds."<<endl;




        }
        cout<<endl;
    }

}
```

1.4.4 Lagrange2D.cpp

```
//
//  Lagrange2D.cpp
//  project3
//
//  Created by Arslan Memon on 10/22/15.
//  Copyright (c) 2015 Arslan. All rights reserved.
//

#include <stdio.h>
#include <stdlib.h>
#include <stdio.h>
```

```
#include <iostream>
#include <math.h>
#include "matrix.hpp"

using namespace std;
double Lagrange_basis(Matrix& x, int i, double z);
double Lagrange2D(Matrix& x, Matrix& y, Matrix& f, double a, double b)
{
    double lagAnswer=0.0;

    for(int j=0;j<y.Size();j++)//loop to evaluate f(a,b).
    //reason for using j first is to efficiently access matrix how it's stored
    {

        for (int i=0; i<x.Size(); i++)
        {
            lagAnswer+=f(i,j)*Lagrange_basis(x,i, a)*Lagrange_basis(y, j, b);
            //multiplying f(i,j) by the lagrange basis for b
            //with nodes y* lagrange basis for a with nodes
            //x to get an approximate of f(a,b) using lagrange interpolation
         }
    }
    return lagAnswer;

}
```

1.4.5 Runge_uniform.cpp

```
//
//  Runge_uniform.cpp
//  project3
//
//  Created by Arslan Memon on 10/24/15.
//  Copyright (c) 2015 Arslan. All rights reserved.
//
#include <stdio.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <math.h>
#include "matrix.hpp"
using namespace std;
double Lagrange2D(Matrix& x, Matrix& y, Matrix& f, double a, double b);
int main()
{
    //runge fuction
    auto funct = [](const double x,const double y) -> double {
        return (1.0/(1+x*x+y*y));
    };
    //set m and n to 6
    int n=6;
    int m=6;
    //set x to m+1 evenly spaced nodes from -4 to 4 and y to
    //n+1 evenly spaced nodes between -4 and 4
```

```
Matrix x=Linspace(-4.0, 4.0, m+1, 1);
Matrix y=Linspace(-4.0, 4.0, n+1, 1);
Matrix f(m+1,n+1);//2d matrix f m+1 by n+1
for(int j=0;j<=n;j++)// use runge function to initialize f
{
    for (int i=0; i<=m;i++) {
        f(i,j)=funct(x(i),y(j));
    }
}

Matrix a=Linspace(-4.0, 4.0, 201, 1);//a is a matrix of 201
//evenly spaced nodes between -4 and 4
Matrix b=Linspace(-4.0, 4.0, 101, 1);//b is a matrix of 101
//evenly spaced nodes between -4 and 4
a.Write("avals.txt");//store a and b
b.Write("bvals.txt");
Matrix p6(201,101);
//evaluate and store f(a,b) using Lagrange2D interpolation and write to disk
for(int j=0;j<101;j++)
{
    for (int i=0; i<201;i++) {
        p6(i,j)=Lagrange2D(x, y, f, a(i), b(j));
    }
}
p6.Write("p6_uni.txt");
//same as above except wth m=n=24
m=24;
n=24;
x=Linspace(-4.0, 4.0, m+1, 1);
y=Linspace(-4.0, 4.0, n+1, 1);
f=Matrix(m+1,n+1);
for(int j=0;j<=n;j++)
{
    for (int i=0; i<=m;i++) {
        f(i,j)=funct(x(i),y(j));
    }
}

Matrix p24(201,101);
for(int j=0;j<101;j++)
{
    for (int i=0; i<201;i++) {
        p24(i,j)=Lagrange2D(x, y, f, a(i), b(j));
    }
}
p24.Write("p24_uni.txt");
Matrix Runge(201,101);
for(int j=0;j<101;j++)
{
    for (int i=0; i<201;i++) {
        Runge(i,j)=funct(a(i),b(j));
    }
}
Runge.Write("Runge.txt");
```

```
}
```

1.4.6 Runge_Chebyshev.cpp

```
//
//  Runge_Chebyshev.cpp
//  project3
//
//  Created by Arslan Memon on 10/24/15.
//  Copyright (c) 2015 Arslan. All rights reserved.
//

#include <stdio.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <math.h>
#include "matrix.hpp"
using namespace std;
double Lagrange2D(Matrix& x, Matrix& y, Matrix& f, double a, double b);
//same as Runge_uniform, except using Chebyshev nodes instead of uniform nodes
int main()
{

    auto funct = [](const double x,const double y) -> double {
        return (1.0/(1+x*x+y*y));
    };
    int n=6;
    int m=6;
    Matrix x(m+1);
    Matrix y(n+1);
    for (int i=0;i<=n;i++)//Chebyshev nodes formula
    {
        x(i)=(4*cos(((2*i+1)*M_PI)/(2*m+2)));
        y(i)=(4*cos(((2*i+1)*M_PI)/(2*m+2)));
    }
    Matrix f(m+1,n+1);
    for(int j=0;j<=n;j++)
    {
        for (int i=0; i<=m;i++) {
            f(i,j)=funct(x(i),y(j));
        }
    }

    Matrix a=Linspace(-4, 4, 201);
    Matrix b=Linspace(-4, 4, 101);
    Matrix p6(201,101);
    for(int j=0;j<101;j++)
    {
        for (int i=0; i<201;i++) {
            p6(i,j)=Lagrange2D(x, y, f, a(i), b(j));
        }
    }
    p6.Write("p6_Cheb.txt");
```

```
    m=24;
    n=24;
    x=Matrix(m+1);
    y=Matrix(n+1);
    f=Matrix(m+1,n+1);
    for (int i=0;i<=n;i++)
    {
        x(i)=(4*cos(((2*i+1)*M_PI)/(2*m+2)));
        y(i)=(4*cos(((2*i+1)*M_PI)/(2*m+2)));
    }

    for(int j=0;j<=n;j++)
    {
        for (int i=0; i<=n;i++) {
            f(i,j)=funct(x(i),y(j));
        }
    }

    Matrix p24(201,101);
    for(int j=0;j<101;j++)
    {
        for (int i=0; i<201;i++) {
            p24(i,j)=Lagrange2D(x, y, f, a(i), b(j));
        }
    }
    p24.Write("p24_Cheb.txt");

}
```

1.4.7 Makefile

```
CXX = g++
CXXFLAGS = -O2 -std=c++11

# makefile targets
all : test_Lagrange.exe test_Newton.exe compare.exe test_Lagrange2D.exe Runge_uni.exe Runge_cheb.exe

test_Lagrange.exe:  Lagrange.cpp test_Lagrange.cpp matrix.o
    $(CXX) $(CXXFLAGS) $^ -o $@

test_Newton.exe : test_Newton.cpp matrix.o Newton_interpolant.o
    $(CXX) $(CXXFLAGS) $^ -o $@

compare.exe : matrix.o Newton_interpolant.o Lagrange.cpp compare.cpp
    $(CXX) $(CXXFLAGS) $^ -o $@

test_Lagrange2D.exe: matrix.o lagrange.cpp test_Lagrange2D.cpp Lagrange2D.o
    $(CXX) $(CXXFLAGS) $^ -o $@

Runge_uni.exe: Runge_uniform.cpp matrix.o lagrange.cpp  Lagrange2D.o
    $(CXX) $(CXXFLAGS) $^ -o $@

Runge_cheb.exe: Runge_Chebyshev.cpp matrix.o lagrange.cpp Lagrange2D.o
    $(CXX) $(CXXFLAGS) $^ -o $@
```

```
Newton_interpolant.o:Newton_interpolant.cpp
    $(CXX) -Wall -g -c $^ -o $@

Lagrange2D.o: Lagrange2D.cpp
     $(CXX) -Wall -g -c $^ -o $@

matrix.o: matrix.cpp
    $(CXX) -Wall -g -c $^ -o $@

clean :
    \rm -f *.o *.txt

realclean : clean
    \rm -f *.exe *~

####### End of Makefile #######
```

In [ ]: