Project 3

Stefan Popov

Oct 25th 2016

MATH3316/CSE3366

0. Building the project.

This project comes with a provided Makefile, as specified in the instructions. To build all executables for this project, simply use the command

make

in Terminal or Command Prompt.

The executable

test_newton.exe

would need to be manually executed to create the required files for this project.

This executable would create the needed files for part 1. of this project, that would later be plotted in this Jupyter Notebook.

To test the validity of the Lagrange2D() method required in part 2., run the executable

test_Lagrange.exe

which was provided in the project description.

Runge_uniform.exe

would create all required files for the plots in part 3.1.

```
Runge_Chebyshev.exe
```

would create all required files for the plots in part 3.2.

To delete all output files simply use the command

```
make clean
```

To delete all output files and executables, type in

```
make realclean
```

1. Newton Interpolation

In the first part of this project, we were to construct an interpolating polynomial in Newton form. To do so, we were given the following specifications :

When asked to interpolate $n+1$ distinct data points $(x_0,y_0),(x_1,y_1), ... , (x_n,y_n)$ by a polynomial of formal degree n using Newton form as

$$p_n(x)= a_0 + a_1(x-x_0)+ a_2(x-x_0)(x-x_1) + ... + a_n(x-x_0)(x-x_1)...(x-x_{n-1})$$

We would take the construction of Newton form to be the process for determining the n+1 unknown coefficients $a_0,a_1, ..., a_n$ using the $n+1$ interpolating conditions $p_n(x_0)=y_0, p_n(x_1)=y_1, ... , p_n(x_n)=y_n$. If one more data point (x_{n+1},y_{n+1}) is added to the previous $n+1$ data points, the new interpolating polynomial of degree $n+1$ that interpolates the expanded data set can be written as

$$p_{n+1}(x)=p_n(x)+ a_{n+1}(x-x_0)(x-x_1)...(x-x_n).$$

The extra coefficient a_{n+1} is given by $$a_{n+1}= \frac{y_{n+1} - p_n(x_{n+1})}{(x_{n+1}-x_0)(x_{n+1}-x_1)...(x_{n+1} - x_n)}$$

providing a recursive formula to find the coefficients $a_1,a_2,...$ by starting with $p_0(x)=a_0=y_0$ and letting n take tha values $0,1,...$ in the above formula.

When using the above recursion we would need to acquire $p_n(x_{n+1})$, which we would do by using nested multiplication. The product $(x_{n+1} - x_0)(x_{n+1}-x_1)...(x_{n+1}-x_n)$ by evaluatin the Newton's basis function

$$\phi_{n+1}(x)=(x-x_0)(x-x_1)...(x-x_n)$$

We would write three C++ functions to fulfill the above :

In [ ]:
```
    double Newton_basis(Matrix& nodes, int n, double x)
```

which purpose is to evaluate the Newton basis function $\phi_{n+1}(x)$. The nodes are passed to the function as a matrix object, as well as the value of x and it returns the evaluation.

In [ ]:
```
    double Newton_nestedform(Matrix& a, Matrix& xnodes,double x)
```

which would evaluate the Newton form $p_n(x)$ using nested multiplication. It would take the coefficients in a Matrix a as a parameter, along with the x nodes and the x used to evaluate. The function would return the evaluation $p_n(x)$.

In [ ]:
```
    Matrix Newton_coefficients(Matrix& xnodes, Matrix& ynodes)
```

This function constructs the Newton form of an interpolating polynomial. The two input matrices contain the x-values and y-values of the data points $(x_0,y_0),(x_1,y_1)...$ to be used by the function. The output matrix contains the coefficients $a_0,a_1,...$ for the Newton form.

Finally, we would write a C++ file test_Newtonform.cpp to test those functions by interpolating the data $(x_i,y_i) (0<=i<=4)$ generated by the polynomial $f(x)= 3.1x^4+2.3x^3-6.6x^2+8.7x+7.9$, where the nodes x_i are -2,-1,0,1,2 and $y_i = f(x_i)$.

After constructing the interpolating polynomial $p_4(x)$, we would plot it along with the function $f(x)$, by evaluating them both at 201 equally spaced x-values on the interval [-3,3]. In another figure we would plot the error $f(x) - p_4(x)$ using the same 201 points. Theoretically, we would expect $p_4(x)$ and $f(x)$ to be very similar, even though we do not take a lot of interpolating nodes, because the interpolating polynomial goes through the data points defined and the error we get should be very small, approaching 0 and no greater than 1.

The following code was used to construct and evaluate the interpolating polynomial $p_4(x)$. In the code we would start by declaring the data points and evaluating the coefficients matrix for them. We then move on to calculating the interpolation matrix, the actual $f(x)$ values and the error, as described in the comments.

In [ ]:

```cpp
/*
MATH3316 Project3
Author: Stefan Popov
Oct 23rd 2016
*/
#include <iostream>
#include "matrix.hpp"
#include <vector>
#include "fcn.hpp"
using namespace std;
vector<double> as;

class fcn : public Fcn {
public:
  double operator()(double x) {   // function evaluation
    return (3.1*x*x*x*x + 2.3*x*x*x - 6.6*x*x + 8.7*x + 7.9);
  }
};


double Newton_basis(Matrix& xnodes, int n, double x);
double Newton_nestedform(Matrix& a, Matrix& xnodes, double x);
Matrix Newton_coefficients(Matrix& xnodes, Matrix& ynodes);


int main(){
    //declare f
    fcn f;
    //declare xi vector
    vector<double> xi= {-2,-1,0,1,2};
    //declare and evaluate yi
    vector<double> yi;
    for ( int i=0; i<5; i++){
        yi.push_back( f(xi[i]));
    }
    //construct matrix with the above vectors
    Matrix xis(xi);
    Matrix yis(yi);

    //construct matrix of coefficients by calling the Newton_coefficients met
hod
    Matrix aMatrix= Newton_coefficients(xis,yis);

    //construct the xValues by creating a Matrix of equally spaced doubles fr
om -3 to 3
    Matrix xValues= Linspace(-3,3,1,201);

    //declare vectors
    vector<double> xInterpolated;
    vector<double> fxs;
    vector<double> errorVector;

    //fill the xInterpolated vector with corresponding value and
    //fill f(x) vector and the error vector accordingly
    for ( int i=0 ;i < xValues.Cols();i++){
        double xInter =Newton_nestedform(aMatrix,xis,xValues[i][0]);
        xInterpolated.push_back(xInter);
        double fx = f(xValues[i][0]);
        fxs.push_back(fx);
        errorVector.push_back(fx-xInter);
    }

    //convert vectors to matrices and write to file
    Matrix xInterpol(xInterpolated);
    Matrix fxMatrix(fxs);
    Matrix errorMatrix(errorVector);
    xInterpol.Write("xInterpol.txt");
    fxMatrix.Write("y.txt");
    errorMatrix.Write("error.txt");
    xValues.Write("x.txt");
    return 0;
}

//Newton_basis function accepts the xNodes, an integer n and
//the double x. It returns the newton basis by simply iterating
//over the xNodes matrix and evaluating at each point
double Newton_basis(Matrix& xnodes, int n, double x){
    double phi= 1.0;
    for(int i=0 ; i < n; i++){
        phi*= (x-xnodes[0][i]);
    }
    return phi;
}

//Newton_nestedform accepts the coefficients matrix a, the
//x nodes matrix and a double x. It returns the evaluation of p_n(x)
//by calling the Newton_basis function and multiplying by the
//corresponding coefficient in the a matrix
double Newton_nestedform(Matrix& a, Matrix& xnodes, double x){
    double pnOfX=0.0;

    for(int i= 0 ; i< a.Rows(); i++){
        pnOfX+=Newton_basis(xnodes,i, x)*a[0][i];
    }
    return pnOfX;
}
//Newton_coefficients method accepts the x and y nodes and returns the update
s the coefficients matrix
Matrix Newton_coefficients(Matrix& xnodes, Matrix& ynodes){
    for(int i = 0; i< xnodes.Rows(); i++){
        Matrix asTemp(as);
        double firstPart = ynodes[0][i] - Newton_nestedform(asTemp,xnodes, xn
odes[0][i]);
        double secondPart =(Newton_basis(xnodes,i,xnodes[0][i]));
        as.push_back((firstPart)/secondPart);
    }
    Matrix aMatrix(as);
    return aMatrix;
}
```

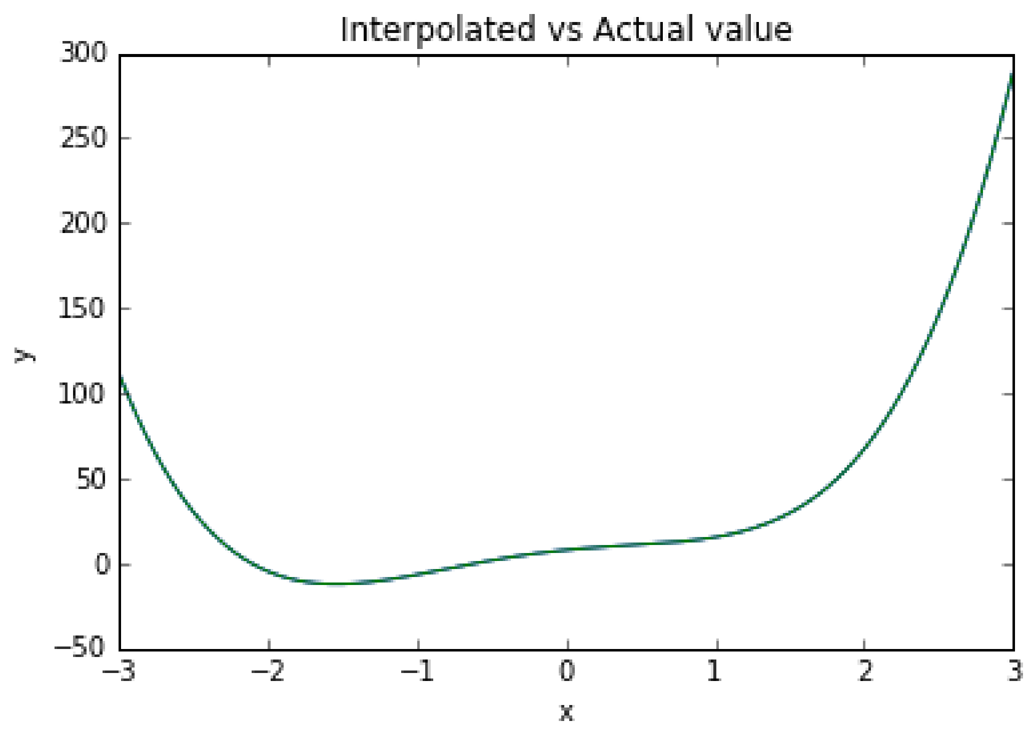
The code produced the following plots when loaded into our jupyter notebook.

In [4]:
```
%pylab inline
error = loadtxt('error.txt')
x = loadtxt('x.txt')
xInterpol= loadtxt('xInterpol.txt')
y = loadtxt('y.txt')
```

```
Populating the interactive namespace from numpy and matplotlib
```

In [5]:
```
plot(x,y)
xlabel('x')
ylabel('y')
title('Interpolated vs Actual value')
plot(x, xInterpol)
```

Out[5]:
```
[<matplotlib.lines.Line2D at 0x7fb88c892cf8>]
```

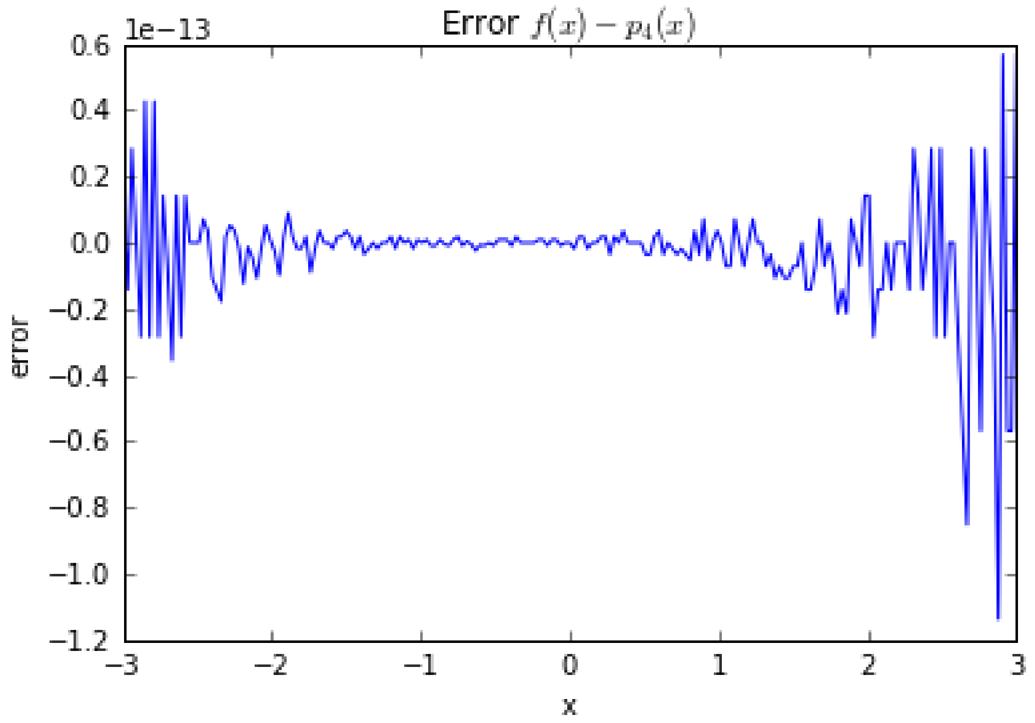


As expected, the interpolating polynomial $p_4(x)$ produces very similar results to what is presented by $f(x)$. This is because the polynomial was built to go through the same data points and only slight errors might occur at times. The plots are virtually the same and any big changes or fluctuations cannot be noticed just by looking at this graph. To make a better assessment, we would need to take a look at the error this polynomial yields by plotting it.

In [9]:
```
plot(x,error)

xlabel('x')
ylabel('error')
title('Error $f(x)-p_4(x)$')
```

Out[9]:
```
<matplotlib.text.Text at 0x7fb889a3f3c8>
```



The plot of the error $f(x) - p_4(x)$ shows us that there are varying errors that mainly occur around the edges of our data set, with much greater oscillation at the positive end than there is at the negative end. This could have been affected by several factors, such as our choice of interpolating nodes, the size of the data set and the function we are trying to interpolate. The function could be causing these, because of great increases around those edges. To test this, we take the plot of the function from another plotting system :

We can easily how the function grows exponentially around the edges of our interval, which is the most probable cause of the worsening of our interpolating polynomial's quality. Over a bigger interval, we would get even bigger oscilattins in the error of the interpolating polynomial.

2. Multi-dimensional interpolation

After being provided a C++ function for evaluating the polynomial interpolant of a set of data points using the Lagrange basis, we were to carry out multi-dimensional interpolation, using the simplest approach, when the data is regularly-spaced over a rectangle and the interpolant is created as a multi-dimensional version of the Lagrange interpolating polynomial. By considering the 1D Lagrange basis functions provided, we may construct a set of 2D Lagrange basis functions via the product of the two bases

$$l_{i,j}(x,y)= \phi_i(x)\psi(y)$$
where $i=1,...,m$ and $j=1,...,n$.

Our 2D Lagrange interpolant for an arbitrary function $f(x,y)$ at the nodes (x_i,y_j) may be computed at a point (a,b) as $$p(a,b)=\sum_{i=0}^{m}\sum_{j=0}^nf(x_i,y_j)l_{i,j}(a,b)$$
We would construct the C++ function Lagrange2D with the following signature

double Lagrange2D(Matrix& x, Matrix& y, Matrix& f, double a, double b);

where x is the input vector of length $m+1$, y is an input vector of length $n+1$, f is an input matrix in $R^{(m+1)×(n+1)}$ that holds the function values $f(x_i,y_j)$, and the doubles a and

b correspond to an evaluation point in the $x-y$ plane. The function evaluates and returns $p(a,b)$. The function was tested with the provided test routine in the file **test_Lagrange2D.cpp** along with the jupyter notebook plot_Lagrange2D.ipynb.

In the Jupyter notebook provided, the python script in the notebook printed the following to the screen, which shows us that the error decreases as we increase the interpolating nodes, which was expected for this function :

In [ ]:
```
p10 success!  ||e10|| =  0.0575277609593  is below tolerance of  0.058
p20 success!  ||e20|| =  3.69151159696e-08  is below tolerance of  3.7e-08
```

The plot with the higher number of interpolating nodes shows a better resemblance of the original one, and the error oscillates less around the edges, compared to the one with less nodes.

To achieve this, the following C++ Code was written in a file called **Lagrange2D.cpp** and compiled with the Makefile:

In [ ]:
```
/*
MATH3316 Project3
Author: Stefan Popov
Oct 23rd 2016
*/
#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include <math.h>
#include "matrix.hpp"

using namespace std;

//Lagrange2D evaluates the 2D Lagrange given the input vectors x,y,z and coef
ficients a and b

double Lagrange2D(Matrix& x, Matrix& y, Matrix& z, double a, double b){
  double result = 0.0;

  for( int i=0; i< x.Rows(); i++){
    for(int j=0; j<y.Rows(); j++){
        result += z(j, i) * Lagrange_basis(x, j, a) * Lagrange_basis(y, i, b)
;
    }

  }
  return result;
}
```

As seen, this function iterates over the x and y nodes and appends the product of the result of the function, the Lagrange basis for x, j and a and the Lagrange basis for y, i and b.

3. The importance of nodes

For this part of the project, we were to consider the two-dimensional Runge function $$f(x,y)= \frac{1}{1+x^2+y^2}$$ in the rectangle $(x,y) \in $ [-4,4]x[-4,4].

3.1 Runge function using uniformly-spaced nodes.

In a single C++ main() routine in a file Runge_uniform.cpp, we were supposed to afccomplish the following tasks:

- Create a set of (m+1) evenly-spaced nodes, x, over the interval [-4,4].
- Create a set of (n+1) evenly-spaced nodes, y, over the interval [-4,4].
- Create a matrix $f\in R^{(m+1)x(n+1)}$ that contains the function values $f(x_i,y_j)$
- Create an array of 201 evenly-spaced evaluation points, a, over the interval [-4,4] and output this to disk as the file **avals.txt**.
- Create an array of 101 evenly-spaced evaluation points, b over the interval [-4,4] and output this to disk as the file **bvals.txt**.
- Use the Lagrange2D() function from the previous section of this project to evaluate the

polynomial interpolant $p(a_i,b_j)$ at teh 201x101 evaluation points, storing the result in the matrices $p_6 \in R^{201x101}$ and $p_{24} \in R^{201x101}$

- Output p_6 and p_{24} to disk to files named **p6_uni.txt** and **p24_uni.txt**
- Lastly, fill a matrix runge $\in R^{201x101}$ with the correct values of $f(a_i,b_j)$ and output this to disk in a file named **Runge.txt**

To fullfil all the above the following C++ code was written in a file named Runge_uniform.cpp. In the code, an interpolate() function was declared to reuse the code for the p_6 and p_{24} polynomials. The function accepts as parameters the integers m and n and the string that contains the name of the corresponding output file and then performs all the above mentioned tasks with the specific m and n provided. Furthermore, the set of x and y nodes were declared as column vectors, which required to overwrite the Lagrange2D() function to iterate over column vectors instead of row vectors. The comments in the code should further explain the steps followed to develop this solution.

In [ ]:
```
/*
MATH3316 Project3
Author: Stefan Popov
Oct 23rd 2016
*/

#include <iostream>
#include <string>
#include "matrix.hpp"
#include "Lagrange.cpp"
using namespace std;


void interpolate(int m, int n, string outputFile);

double Lagrange2D(Matrix& x, Matrix& y, Matrix& z, double a, double b);

int main(){
        //declare runge function

        //declare n = m =6
        int n1=6, m1=6;
        int n2=24, m2=24;

        //call interpolate function
        interpolate(n1,m1, "p6_uni.txt");
        interpolate(n2,m2, "p24_uni.txt");

}

void interpolate(int m, int n, string outputFile){
        //declare f
        auto f = [](const double x, const double y) -> double {
```

```
        return 1/(1+x*x+y*y);
        };

        //create a set of m+1 nodes from -4 to 4
        Matrix xNodes = Linspace(-4,4,1,m+1);

        //create a set of n+1 nodes from -4 to 4
        Matrix yNodes = Linspace(-4,4,1,n+1);

        //create a matrix with dimensions [m+1][n+1] for the function values
f(x_i,y_i)
        Matrix fValues(m+1,n+1);
        //fill values for f matrix
        for(int i=0; i<m+1;i++){
                for(int j=0; j<n+1;j++){
                        fValues[j][i]=f(xNodes[i][0],yNodes[j][0]);
                }
        }



        //create an array of 201 evenly spaced evaluation points
        Matrix aVals = Linspace(-4,4,1,201);
        //write to disk
        aVals.Write("avals.txt");

        //create an array of 101 evenly spaced evaluation points
        Matrix bVals = Linspace(-4,4,1,101);
        //write to disk
        bVals.Write("bvals.txt");

        fValues.Write("zVals.txt");

        Matrix p6(201,101);
        //evaluate the polynomial interpolant p(a_i,b_j)
        for( int i = 0 ; i < aVals.Cols(); i++){
                for ( int j=0 ; j<bVals.Cols(); j++){
                        p6(i,j) = Lagrange2D(xNodes, yNodes, fValues, aVals[i
][0], bVals[j][0]);
                }
        }

        //find f(a,b)
        Matrix Runge(201,101);
        for( int i = 0 ; i < aVals.Cols(); i++){
                for ( int j=0 ; j<bVals.Cols(); j++){
                        Runge[j][i] = f(aVals[i][0],bVals[j][0]);
                }
        }
        //write to files
        Runge.Write("Runge.txt");

        p6.Write(outputFile.c_str());

}
//lagrange2D function evaluator
double Lagrange2D(Matrix& x, Matrix& y, Matrix& z, double a, double b){
  double result = 0.0;
  for( int i=0; i< x.Cols(); i++){
    for(int j=0; j<y.Cols(); j++){
        result += z(i, j) * Lagrange_basis(x, i, a) * Lagrange_basis(y, j, b)
;
    }

  }
  return result;
```

```
}
```

3.2 Runge function using Chebyshev nodes

In this part, instead of using uniformly-spaced nodes we were to use Chebyshev nodes, given by the formula for (m+1) Chebyshev nodes over the interval [-L,L]
$$x_i=Lcos{\frac{(2i+1)\pi}{2m+2}}$$
for $i=0,...,m$

In a single C++ main() routine in a file **Runge_Chebyshev.cpp** we would repeat the steps in 3.1, but using the output files **p6_Cheb.txt** and **p24_Cheb.txt**.

Similarly, to fullfil the requirements for Chebyshev nodes the following C++ code was written in a file named Runge_Chebyshev.cpp. In the code, an interpolate() function was declared to reuse the code for the p_6 and p_{24} polynomials. The function accepts as parameters the integers m and n and the string that contains the name of the corresponding output file and then performs all the above mentioned tasks with the specific m and n provided. Furthermore, the set of x and y nodes were declared as column vectors, which required to overwrite the Lagrange2D() function to iterate over column vectors instead of row vectors. The comments in the code should further explain the steps followed to develop this solution.

In [ ]:
```
/*
MATH3316 Project3
Author: Stefan Popov
Oct 23rd 2016
*/

#include <iostream>
#include <string>
#include <math.h>
#include "matrix.hpp"
#include "Lagrange.cpp"
using namespace std;


void interpolate(int m, int n, string outputFile);

double Lagrange2D(Matrix& x, Matrix& y, Matrix& z, double a, double b);

int main(){
        //declare runge function

        //declare n = m =6
        int n1=6, m1=6;
        int n2=24, m2=24;

        //call interpolate function
        interpolate(n1,m1, "p6_Cheb.txt");
        interpolate(n2,m2, "p24_Cheb.txt");

}


//evaluate chebyshevNodes
```

```
Matrix chebyshevNodesBuild(int m){
        Matrix x(1,m+1);
        for(int i =0 ;i <m+1; i++){
                x[i][0]= 4*cos((2*(i+1)*M_PI)/(2*m+2));
        }
        return x;
}

void interpolate(int m, int n, string outputFile){
        //declare function
        auto f = [](const double x, const double y) -> double {
        return 1/(1+x*x+y*y);
        };

        //create a set of m+1 chebyshev
        Matrix xNodes = chebyshevNodesBuild(m);

        //create a set of n+1 chebyshev nodes
        Matrix yNodes = chebyshevNodesBuild(n);

        //create a matrix with dimensions [m+1][n+1] for the function values
f(x_i,y_i)
        Matrix fValues(m+1,n+1);
        //fill values for f matrix
        for(int i=0; i<m+1;i++){
                for(int j=0; j<n+1;j++){
                        fValues[j][i]=f(xNodes[i][0],yNodes[j][0]);
                }
        }


        //create an array of 201 evenly spaced evaluation points
        Matrix aVals = Linspace(-4,4,1,201);
        //write to disk
        aVals.Write("avals.txt");

        //create an array of 101 evenly spaced evaluation points
        Matrix bVals = Linspace(-4,4,1,101);
        //write to disk
        bVals.Write("bvals.txt");



        Matrix p6(201,101);
        //evaluate the polynomial interpolant p(a_i,b_j)
        for( int i = 0 ; i < aVals.Cols(); i++){
                for ( int j=0 ; j<bVals.Cols(); j++){
                        p6[j][i] = Lagrange2D(xNodes, yNodes, fValues, aVals[
i][0], bVals[j][0]);
                }
        }


        //write to file
        p6.Write(outputFile.c_str());

}
double Lagrange2D(Matrix& x, Matrix& y, Matrix& z, double a, double b){
  double result = 0.0;

  for( int i=0; i< x.Cols(); i++){
    for(int j=0; j<y.Cols(); j++){
        result += z(j, i) * Lagrange_basis(x, j, a) * Lagrange_basis(y, i, b)
;
    }
  }
  return result;

}
```

3.3. Graphing and analysis of results

To examine our results, we would load all created files using the Numpy loadtxt command and plot each surface in separate figure windows.
Afterwards, we would compute the error in each polynomial interpolant $err_{i,j}=|f_{i,j}-p_{i,j}|$ and plot those in separate figure windows.
All the plots and error evaluations were performed in the second separate Jupyter Notebook called Runge2D.ipynb, which can be found in the packet. The following analysis is based on the results displayed in the graphs there and will refer to them.

The first plot presented in the Runge2D jupyter notebook is the function evaluation for the Runge function over the a and b nodes. All other plots will be compared to that specific one and evaluated based on the similarity and error to it.

3.3.1. p_6 for uniformly spaced nodes

As seen in the graph for p_6 for uniformly spaced nodes, there are big fluctuations around the edges of the graph, even though near x=0 and y=0 the graphs look somewhat familiar. This is caused by the fact that we are using very little evaluation nodes, which are uniformly-spaced, thus not providing a perfect estimation for the problem. The error plot $|f(x,y) - p_{6uni}|$ confirms our previous assumption : the graph fluctuates around the edges and has little error around x=0 y=0.

3.3.2. p_{24} for uniformly spaced nodes

The graph for p_{24} shows us the so-called Runge's phenomenon. Even though we increase the number of nodes, since they are uniformly-spaced we experience great oscillation around the edges, which is contrary to what we would have expected when increasing the number of uniformly-spaced nodes over an interval. This does not follow Weierstrass'theorem, because the theorem does not state how to find the interpolating polynomial. The error around the edges is very big and also confirms these oscillations.

3.3.3. p_6 for Chebyshev nodes

For Chebyshev nodes, we notice a lot more similarity between the original and the Chebyshev p_6 polynomial, however, the elevation of the newly graphed polynomial begins a lot earlier than it should, according to the original. The error is comparative to the error presented in the first polynomial graphed in magnitude, however, it appears a lot less around the edges.

3.3.4. p_{24} for Chebyshev nodes

Unlike the graph for uniform nodes, we do not witness Runge's phenonemon, as the Chebyshev nodes are specially built and not uniformly spaced. This graph resembles maximally the original and the error between the two is very low and mainly around x=0 and y=0.

3.3.5 Ranking

In terms of quality, these four interpolants would be ranked in the following order :

1. p_{24} for Chebyshev nodes

2. p_6 for Chebyshev nodes
3. p_6 for uniformly-spaced nodes
4. p_{24} for uniformly-spaced nodes

4. Makefile

The following Makefile was used to achieve all mentioned in 0.

In [ ]:
```
###################################################################
#  Makefile for project 3
#
#  Daniel R. Reynolds
#  SMU Mathematics
#  Math 3316
#  28 September 2015
# Edited by Stefan Popov on Oct 23 2016 for Project 3 in Math 3316
###################################################################

# compiler & flags
CXX = g++
CXXFLAGS = -O2 -std=c++11

# makefile targets
all : test_Lagrange.exe test_Lagrange2D.exe test_newton.exe Runge_Chebyshev.e
xe Runge_uniform.exe

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

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

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

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

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

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

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



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