Project 3 – Polynomial Interpolation

Erik Gabrielsen

MATH 3316

November 8, 2015

Part I – Newton Interpolation

In Part I of this lab, I was to implement Newton's method of interpolating polynomials and compare its performance to that of Lagrange basis. I created a file with the following two methods:

(1) `Matrix Newton_coefficients(Matrix& x, Matrix& y);`
(2) `double Newton_evaluate(Matrix& x, Matrix& c, double z);`

Method (1) takes two vectors both in R^{n+1} and returns the coefficients {c} that define the Newton interpolated Polynomial as demonstrated in the textbook on page 166. I will then use these values to pass into my `Newton_evaluate` method which evaluates a polynomial using the coefficients in the form of a `Matrix c.` I followed the pseudocode on page 166 of the textbook in order to evaluate a given polynomial using nested multiplication to optimize performance. After creating these two methods, I then created a test file `test_Newton.cpp` which evaluates the function data values of cosh($2x^2$) with n nodes evenly spaced on the interval (0, 1). I would use n = 10 and n = 20, to compare the results with the provided `test_Lagrange.cpp` file. These are the results from the newton interpolation test:

```
interpolants and errors using 10 nodes:

    z        f(z)                p(z)                error
    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 20 nodes:

    z        f(z)                p(z)                error
    0.025    1.0000007812501     1.0000007812500     6.3e-14
    0.075    1.0000632819174     1.0000632819174     3.8e-15
    0.125    1.0004883209877     1.0004883209877           0
    0.175    1.0018763677492     1.0018763677492     4.4e-16
    0.225    1.0051301616855     1.0051301616855           0
    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     4.4e-16
    0.475    1.1035527079242    1.1035527079242   6.7e-16
    0.525    1.1558250062399    1.1558250062399   2.2e-16
    0.575    1.2267090054760    1.2267090054760         0
    0.625    1.3210170862936    1.3210170862936   2.2e-16
    0.675    1.4447256306661    1.4447256306661   2.2e-16
    0.725    1.6053630078155    1.6053630078155   2.2e-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   8.9e-15
    0.925    2.8582584261957    2.8582584261957   6.4e-14
    0.975    3.4218194055573    3.4218194055581     8e-13
```

Comparing these results with the Lagrange basis of interpolation (results found in the appendix) I noticed that when using 10 nodes, the results were the exact same. However, when I increase the nodes to 20, the results differ in that Newton's approximation is more accurate than Lagrange for the first two nodes, but then they become relatively the same throughout. I did expect for both tests to result in about the same error and approximations but did not expect for them both to be exactly the same with 10 nodes. After assessing the results even further we can see that both tests show that using more nodes in the approximation decreases the error in the approximation. When using 10 nodes the error is around 10^{-7} but when increased to 20 nodes the error is cut in half to 10^{-16}.

After testing these two evaluation tools for interpolating polynomials, I then compared the two methods performance time by creating a file called `compare.cpp.` In my compare file, I run 16 total tests of each interpolation routines using nodes evenly spaced over the interval [-2, 2] and create a set of function data values, $y = \cosh(x^2/3)$. For testing I used all combinations of nodes n = {10, 20, 40, 80} and evaluation points m = {100, 1000, 10000, 100000}. I then recorded the time it took for each method to evaluate p(z).

```
Time with n = 10 | m = 100
      Newton time:        0.097 ms
      Lagrange time:      0.474 ms
Time with n = 10 | m = 1000
      Newton time:        1.003 ms
      Lagrange time:      4.743 ms
Time with n = 10 | m = 10000
      Newton time:        9.652 ms
      Lagrange time:      51.685 ms
Time with n = 10 | m = 100000
      Newton time:        93.156 ms
      Lagrange time:      476.112 ms
Time with n = 20 | m = 100
      Newton time:        0.423 ms
      Lagrange time:      3.251 ms
Time with n = 20 | m = 1000
      Newton time:        3.077 ms
      Lagrange time:      17.265 ms
```

```
Time with n = 20 | m = 10000
      Newton time:        32.444 ms
      Lagrange time:      175.356 ms
Time with n = 20 | m = 100000
      Newton time:        318.891 ms
      Lagrange time:      1792.06 ms
Time with n = 40 | m = 100
      Newton time:        1.217 ms
      Lagrange time:      6.596 ms
Time with n = 40 | m = 1000
      Newton time:        10.632 ms
      Lagrange time:      80.594 ms
Time with n = 40 | m = 10000
      Newton time:        115.626 ms
      Lagrange time:      673.112 ms
Time with n = 40 | m = 100000
      Newton time:        1139.53 ms
      Lagrange time:      6789.34 ms
Time with n = 80 | m = 100
      Newton time:        4.332 ms
      Lagrange time:      24.897 ms
Time with n = 80 | m = 1000
      Newton time:        45.952 ms
      Lagrange time:      264.176 ms
Time with n = 80 | m = 10000
      Newton time:        429.766 ms
      Lagrange time:      2640.24 ms
Time with n = 80 | m = 100000
      Newton time:        4287.93 ms
 Lagrange time:      27338.2 ms
```

After viewing the results, it is easy to see that Newton's method drastically out-performs Lagrange in evaluating the polynomial p(x) at all combinations of n and m. As the number of evaluation points m increased, I noticed that each time increased x10 which is to be expected as m also increases by x10. The reason that Newton's method will always interpolate faster than Lagrange, is that Newton's method only has to find the newton coefficients once, whereas in Lagrange, the Lagrange basis is found during every single iteration of evaluating each point. This would explain the drastic increase in time for Lagrange as it runs at a time complexity of O(n^2) compared to Newton which runs at approximately O(nlogn).

Part II – Multi-Dimensional Interpolation

In Part II, I will update the Lagrange method to be used in two dimensional polynomial interpolation. To construct Lagrange in 2D I will create a file called `Lagrange2D.cpp` that will contain the following method signature:

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

Within this function, I used the Lagrange basis defined in `Lagrange.cpp` for both x and y portions of the input matrices. Because the this version is 2 dimensional, I not only have to interpolate each data point, which runs in O(n^2) complexity, but I also need to computer the Lagrange basis twice for each point. This in turn adds to the overall complexity of Lagrange2D and would increase the complexity of the overall algorithm to be within O(n^3). Additionally, as you wish to decrease the error by increasing the tolerance, the number of nodes to each level would have to increase, meaning that the number of nodes would increase quadratically and not linearly, making run time increase dramatically.

By analyzing the data after plotting my Lagrange2D interpolation using the provided python files, I can see that the error for most of the Lagrange 2D interpolation is practically non-existent. It is only on the outlying points of the graph where the error raises to be above 0.1. The error is also reduced when the number of nodes is increased from 10 to 20 which is to be expected.

Part III – The Importance of Nodes

In Part III, I will interpolate the Runge Function using uniformly spaced nodes and using strategically chosen nodes, Chebyshev nodes on the interval [-4, 4] using n nodes over x, and m nodes over y. n and m will be set to 6 for one iteration and 24 for the next. I then saved the resulting matrices to disk and uploaded them to a Python file that opens them and plots the points in a 3D graph. The Results are shown in the appendix.

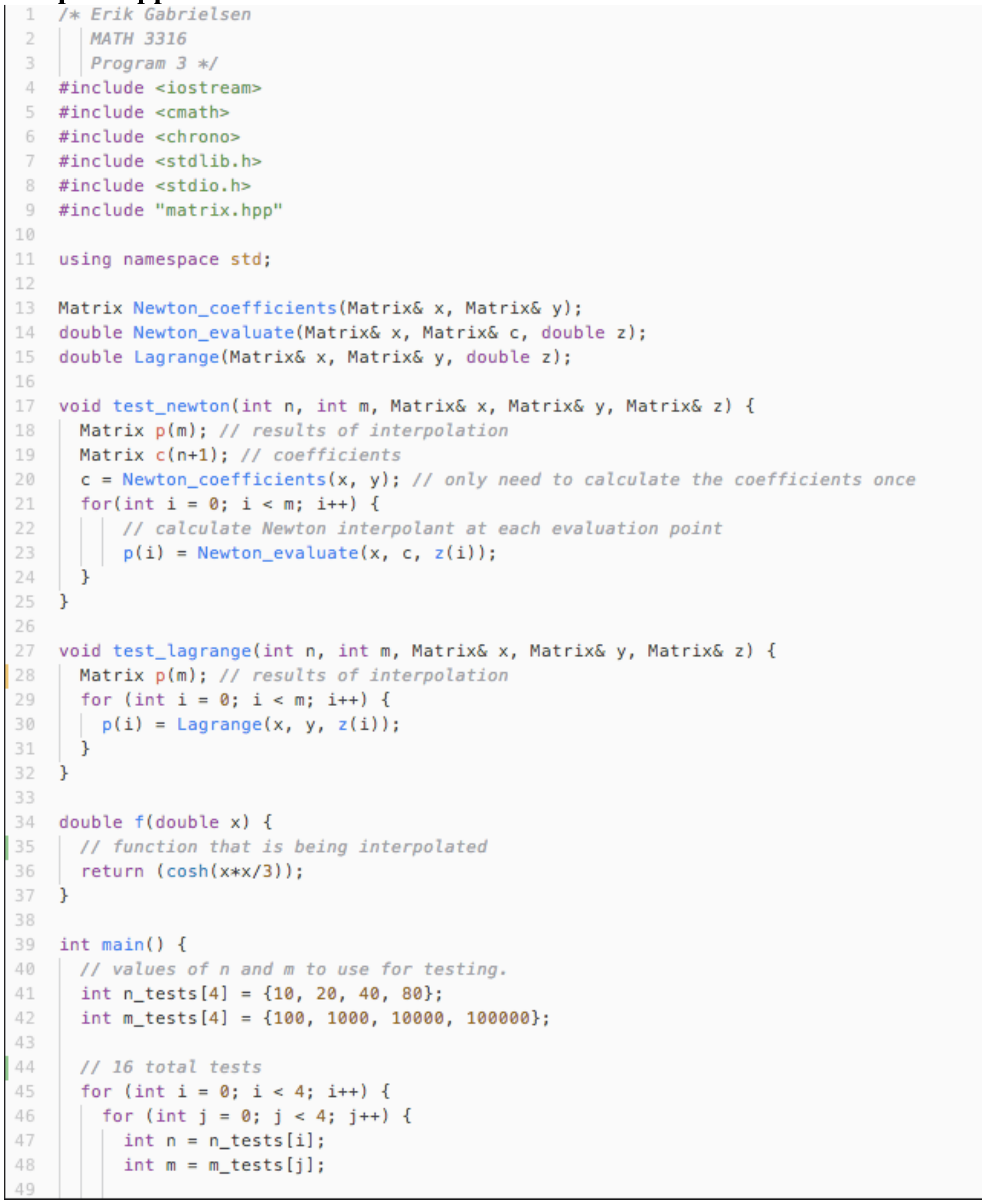
Based on the results, I found that evenly spaced nodes is not always the best way to approximate a function with a polynomial. I would expect that using 24 evenly spaced nodes would be more accurate than by using 6 evenly spaced nodes but it turns out that it is just the opposite. I also discovered that when using Chebyshev's nodes the results are much more accurate then using uniformly spaced nodes. As expected the more nodes that Chebyshev uses the more accurate the approximation is. I ranked each approximation in the following order of quality

1. Chebyshev using 24 nodes
2. Chebyshev using 6 nodes
3. Evenly spaced 6 nodes
4. Evenly spaced 24 nodes

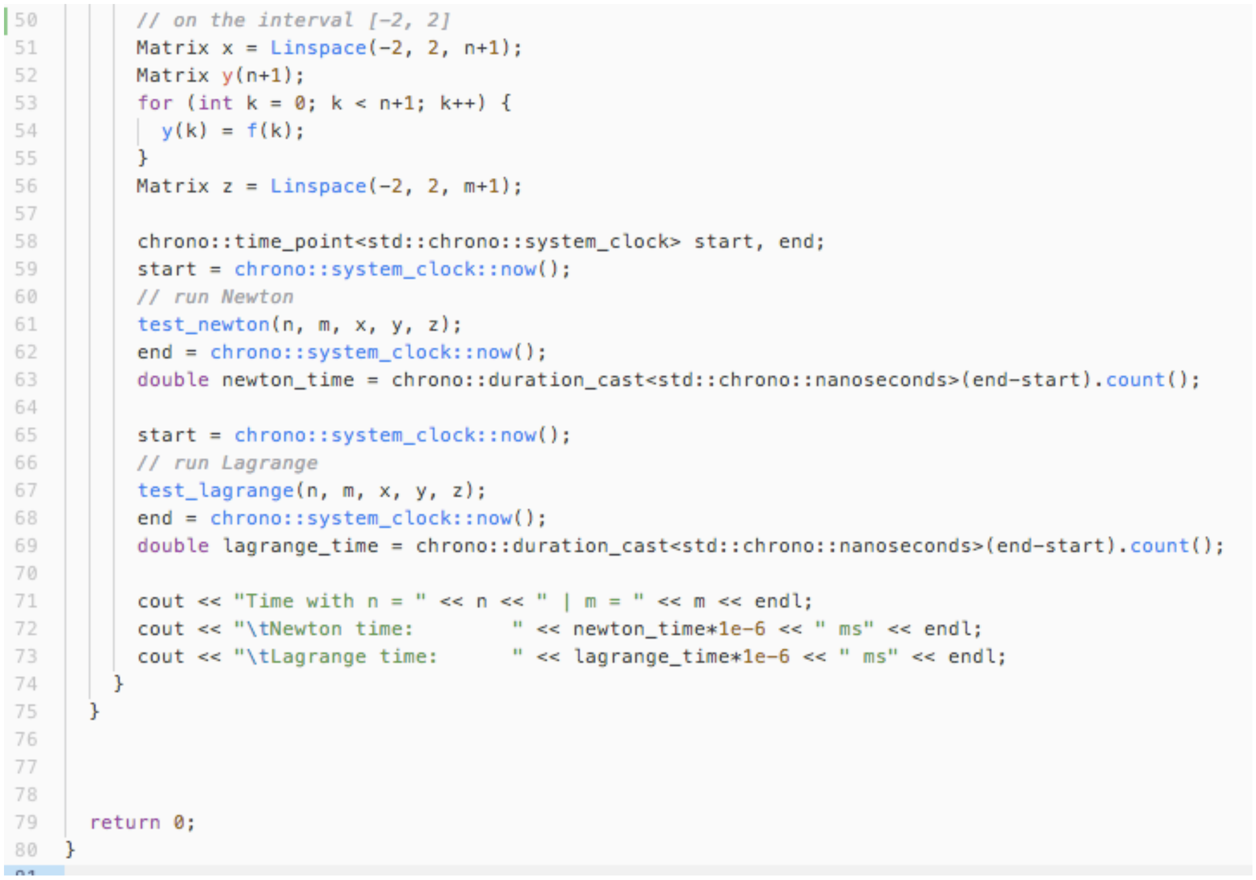
I also noticed that for Chebyshev, the error seemed to be more closer to the heart of the function whereas in uniform spaced nodes the error is extreme towards the perimeter. Overall, it is better to use nodes that are strategically spaced than uniform.

Appendix I – Code

compare.cpp

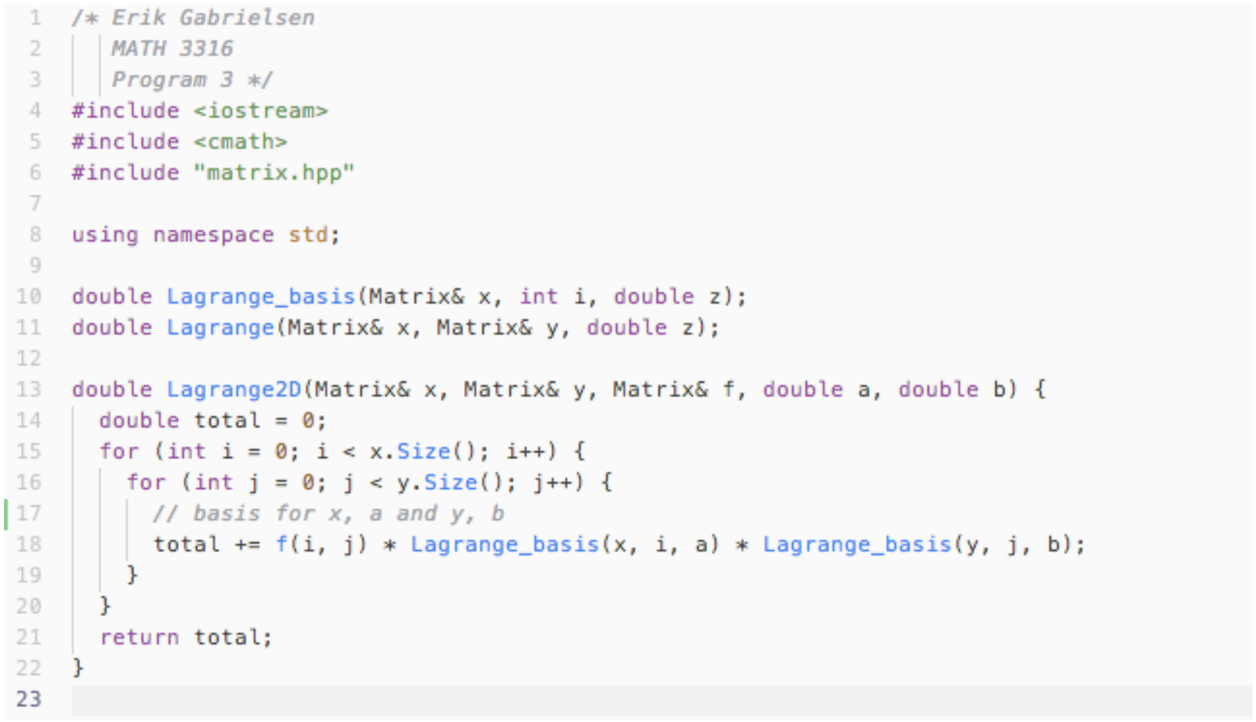


```
/* Erik Gabrielsen
   MATH 3316
   Program 3 */
#include <iostream>
#include <cmath>
#include <chrono>
#include <stdlib.h>
#include <stdio.h>
#include "matrix.hpp"

using namespace std;

Matrix Newton_coefficients(Matrix& x, Matrix& y);
double Newton_evaluate(Matrix& x, Matrix& c, double z);
double Lagrange(Matrix& x, Matrix& y, double z);

void test_newton(int n, int m, Matrix& x, Matrix& y, Matrix& z) {
  Matrix p(m); // results of interpolation
  Matrix c(n+1); // coefficients
  c = Newton_coefficients(x, y); // only need to calculate the coefficients once
  for(int i = 0; i < m; i++) {
      // calculate Newton interpolant at each evaluation point
      p(i) = Newton_evaluate(x, c, z(i));
  }
}

void test_lagrange(int n, int m, Matrix& x, Matrix& y, Matrix& z) {
  Matrix p(m); // results of interpolation
  for (int i = 0; i < m; i++) {
    p(i) = Lagrange(x, y, z(i));
  }
}

double f(double x) {
  // function that is being interpolated
  return (cosh(x*x/3));
}

int main() {
  // values of n and m to use for testing.
  int n_tests[4] = {10, 20, 40, 80};
  int m_tests[4] = {100, 1000, 10000, 100000};

  // 16 total tests
  for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++) {
      int n = n_tests[i];
      int m = m_tests[j];

```



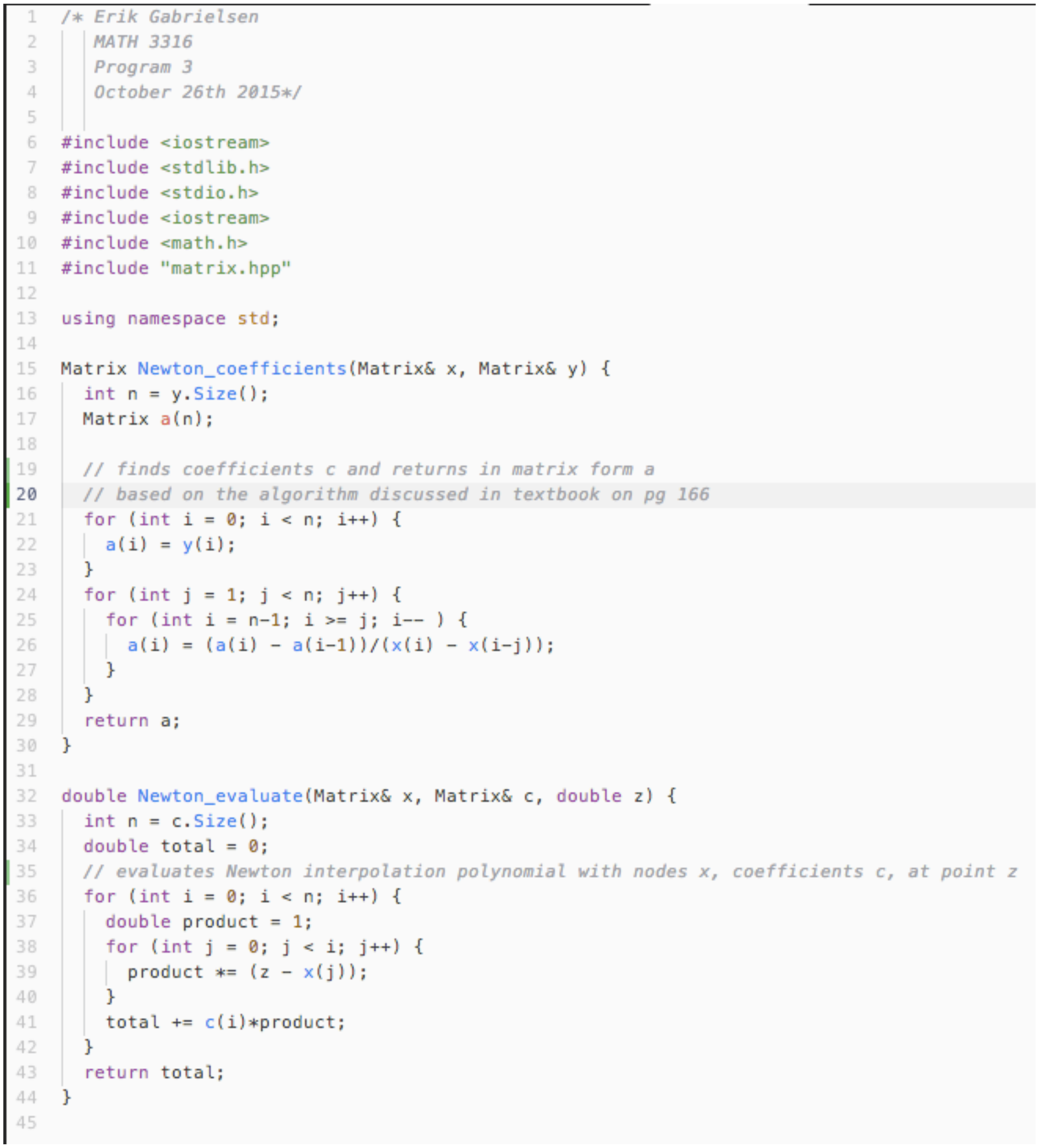
```cpp
      // on the interval [-2, 2]
      Matrix x = Linspace(-2, 2, n+1);
      Matrix y(n+1);
      for (int k = 0; k < n+1; k++) {
        y(k) = f(k);
      }
      Matrix z = Linspace(-2, 2, m+1);

      chrono::time_point<std::chrono::system_clock> start, end;
      start = chrono::system_clock::now();
      // run Newton
      test_newton(n, m, x, y, z);
      end = chrono::system_clock::now();
      double newton_time = chrono::duration_cast<std::chrono::nanoseconds>(end-start).count();

      start = chrono::system_clock::now();
      // run Lagrange
      test_lagrange(n, m, x, y, z);
      end = chrono::system_clock::now();
      double lagrange_time = chrono::duration_cast<std::chrono::nanoseconds>(end-start).count();

      cout << "Time with n = " << n << " | m = " << m << endl;
      cout << "\tNewton time:        " << newton_time*1e-6 << " ms" << endl;
      cout << "\tLagrange time:      " << lagrange_time*1e-6 << " ms" << endl;
    }
  }



  return 0;
}
```

Lagrange2D.cpp



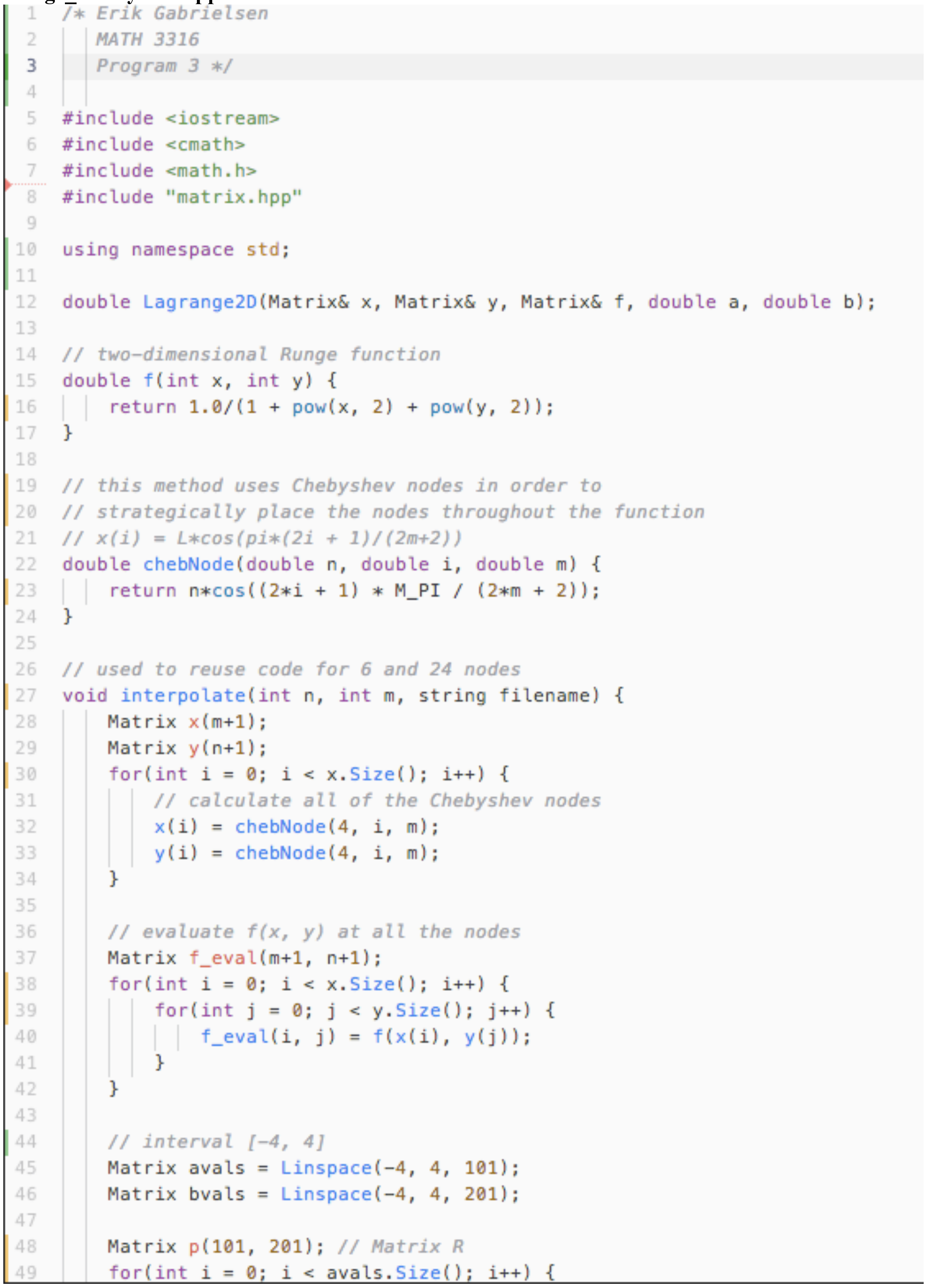
```cpp
/* Erik Gabrielsen
   MATH 3316
   Program 3 */
#include <iostream>
#include <cmath>
#include "matrix.hpp"

using namespace std;

double Lagrange_basis(Matrix& x, int i, double z);
double Lagrange(Matrix& x, Matrix& y, double z);

double Lagrange2D(Matrix& x, Matrix& y, Matrix& f, double a, double b) {
  double total = 0;
  for (int i = 0; i < x.Size(); i++) {
    for (int j = 0; j < y.Size(); j++) {
      // basis for x, a and y, b
      total += f(i, j) * Lagrange_basis(x, i, a) * Lagrange_basis(y, j, b);
    }
  }
  return total;
}
```

Newton_interpolant.cpp

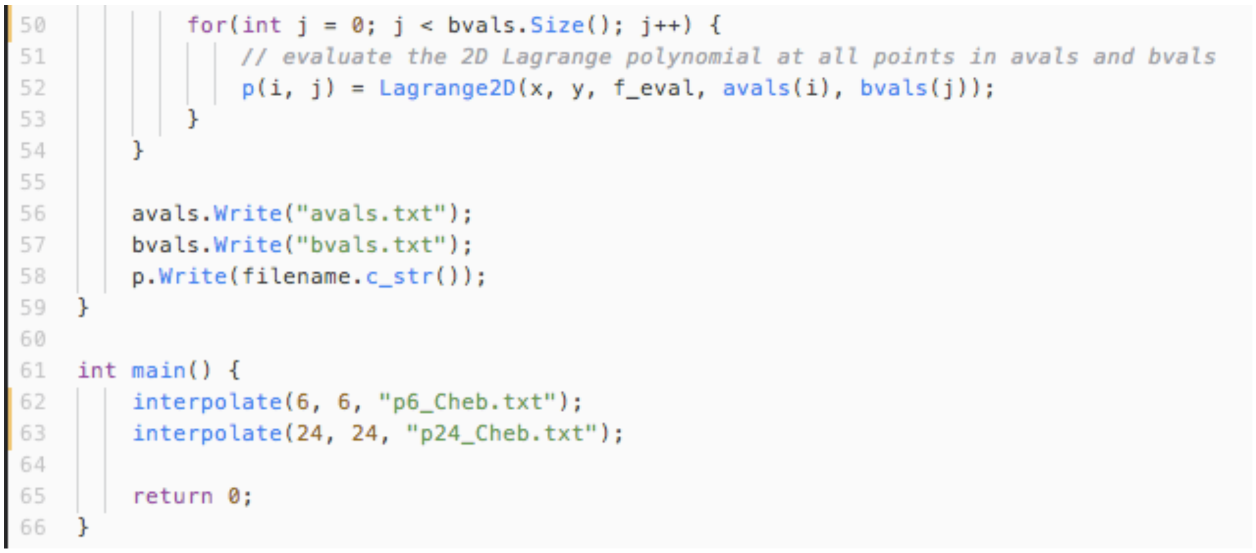


```cpp
/* Erik Gabrielsen
   MATH 3316
   Program 3
   October 26th 2015*/

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

using namespace std;

Matrix Newton_coefficients(Matrix& x, Matrix& y) {
  int n = y.Size();
  Matrix a(n);

  // finds coefficients c and returns in matrix form a
  // based on the algorithm discussed in textbook on pg 166
  for (int i = 0; i < n; i++) {
    a(i) = y(i);
  }
  for (int j = 1; j < n; j++) {
    for (int i = n-1; i >= j; i-- ) {
      a(i) = (a(i) - a(i-1))/(x(i) - x(i-j));
    }
  }
  return a;
}

double Newton_evaluate(Matrix& x, Matrix& c, double z) {
  int n = c.Size();
  double total = 0;
  // evaluates Newton interpolation polynomial with nodes x, coefficients c, at point z
  for (int i = 0; i < n; i++) {
    double product = 1;
    for (int j = 0; j < i; j++) {
      product *= (z - x(j));
    }
    total += c(i)*product;
  }
  return total;
}

```

Runge_Chebyshev.cpp

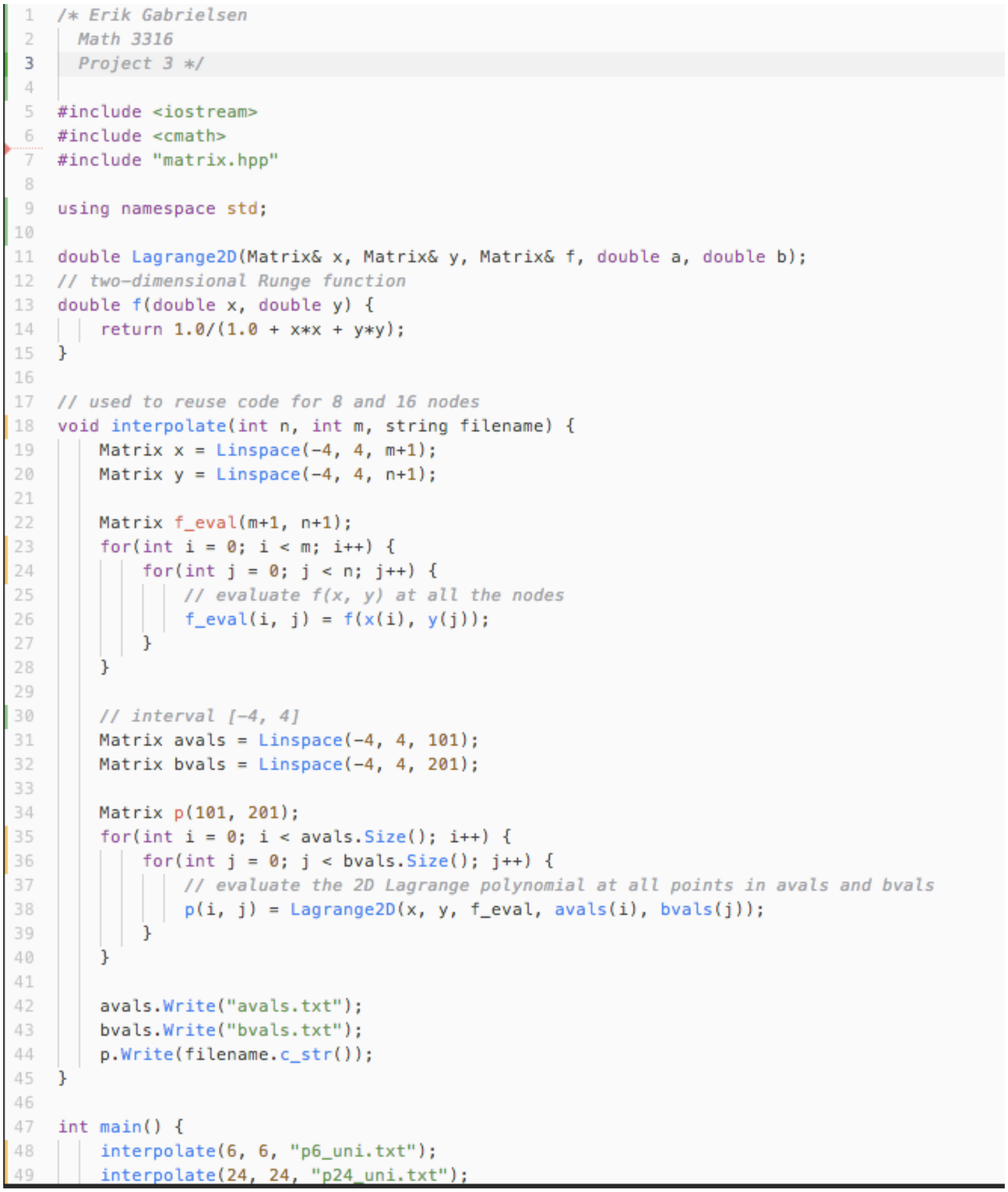


```cpp
/* Erik Gabrielsen
   MATH 3316
   Program 3 */

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

using namespace std;

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

// two-dimensional Runge function
double f(int x, int y) {
    return 1.0/(1 + pow(x, 2) + pow(y, 2));
}

// this method uses Chebyshev nodes in order to
// strategically place the nodes throughout the function
// x(i) = L*cos(pi*(2i + 1)/(2m+2))
double chebNode(double n, double i, double m) {
    return n*cos((2*i + 1) * M_PI / (2*m + 2));
}

// used to reuse code for 6 and 24 nodes
void interpolate(int n, int m, string filename) {
    Matrix x(m+1);
    Matrix y(n+1);
    for(int i = 0; i < x.Size(); i++) {
        // calculate all of the Chebyshev nodes
        x(i) = chebNode(4, i, m);
        y(i) = chebNode(4, i, m);
    }

    // evaluate f(x, y) at all the nodes
    Matrix f_eval(m+1, n+1);
    for(int i = 0; i < x.Size(); i++) {
        for(int j = 0; j < y.Size(); j++) {
            f_eval(i, j) = f(x(i), y(j));
        }
    }

    // interval [-4, 4]
    Matrix avals = Linspace(-4, 4, 101);
    Matrix bvals = Linspace(-4, 4, 201);

    Matrix p(101, 201); // Matrix R
    for(int i = 0; i < avals.Size(); i++) {
```

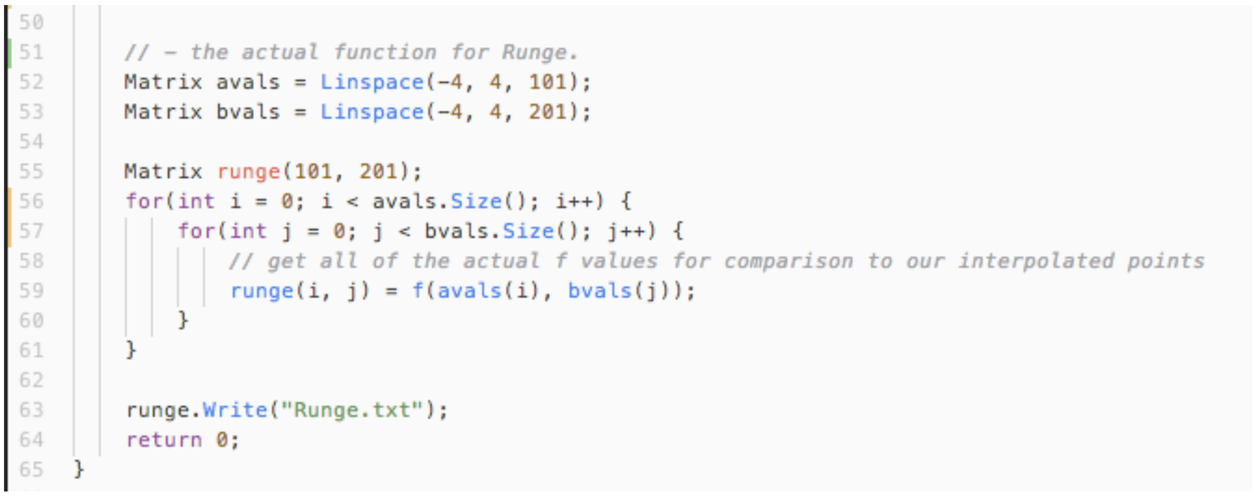


```cpp
        for(int j = 0; j < bvals.Size(); j++) {
            // evaluate the 2D Lagrange polynomial at all points in avals and bvals
            p(i, j) = Lagrange2D(x, y, f_eval, avals(i), bvals(j));
        }
    }

    avals.Write("avals.txt");
    bvals.Write("bvals.txt");
    p.Write(filename.c_str());
}

int main() {
    interpolate(6, 6, "p6_Cheb.txt");
    interpolate(24, 24, "p24_Cheb.txt");

    return 0;
}
```

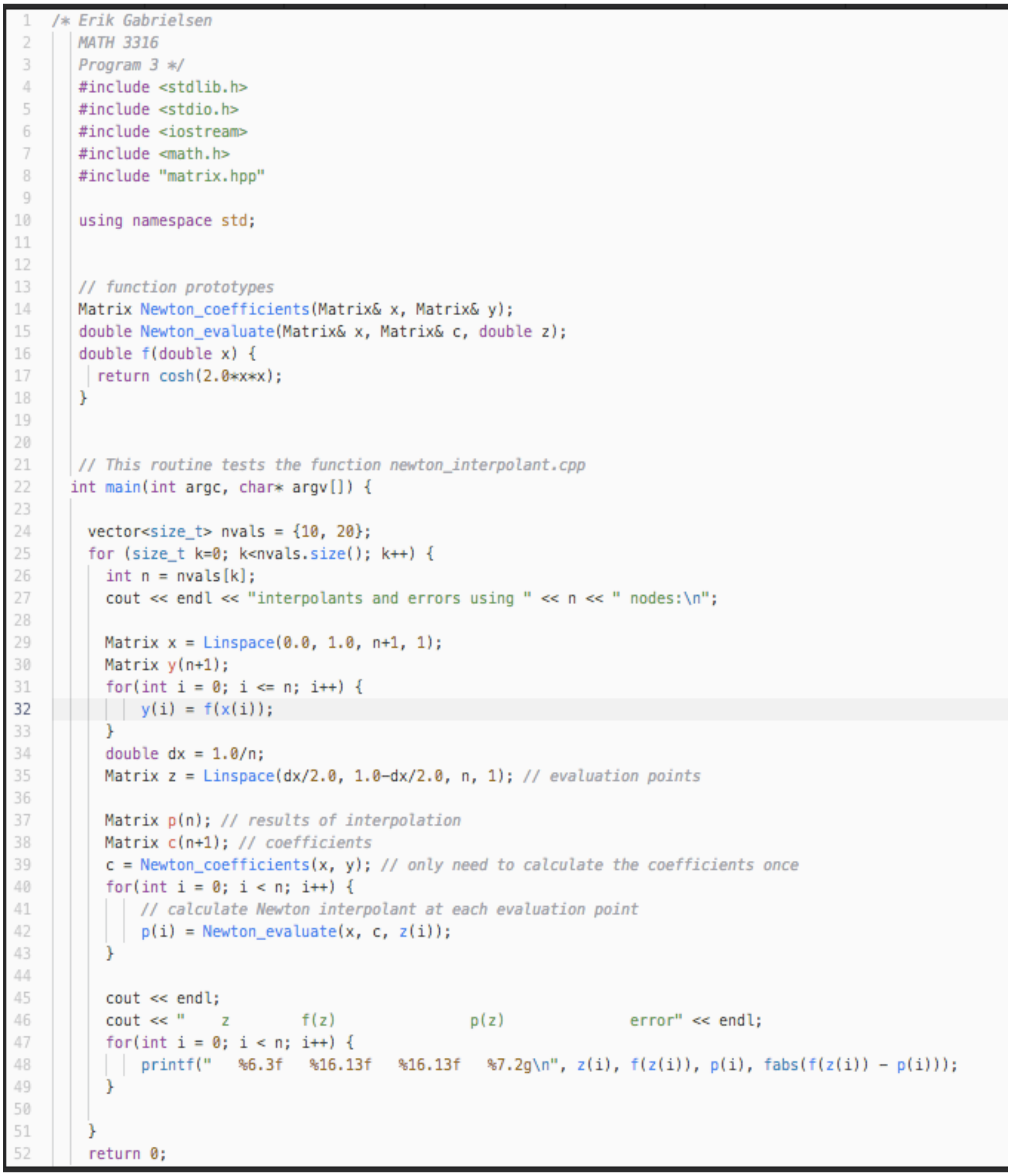
Runge_uniform.cpp



```cpp
/* Erik Gabrielsen
 Math 3316
 Project 3 */

#include <iostream>
#include <cmath>
#include "matrix.hpp"

using namespace std;

double Lagrange2D(Matrix& x, Matrix& y, Matrix& f, double a, double b);
// two-dimensional Runge function
double f(double x, double y) {
    return 1.0/(1.0 + x*x + y*y);
}

// used to reuse code for 8 and 16 nodes
void interpolate(int n, int m, string filename) {
    Matrix x = Linspace(-4, 4, m+1);
    Matrix y = Linspace(-4, 4, n+1);

    Matrix f_eval(m+1, n+1);
    for(int i = 0; i < m; i++) {
        for(int j = 0; j < n; j++) {
            // evaluate f(x, y) at all the nodes
            f_eval(i, j) = f(x(i), y(j));
        }
    }

    // interval [-4, 4]
    Matrix avals = Linspace(-4, 4, 101);
    Matrix bvals = Linspace(-4, 4, 201);

    Matrix p(101, 201);
    for(int i = 0; i < avals.Size(); i++) {
        for(int j = 0; j < bvals.Size(); j++) {
            // evaluate the 2D Lagrange polynomial at all points in avals and bvals
            p(i, j) = Lagrange2D(x, y, f_eval, avals(i), bvals(j));
        }
    }

    avals.Write("avals.txt");
    bvals.Write("bvals.txt");
    p.Write(filename.c_str());
}

int main() {
    interpolate(6, 6, "p6_uni.txt");
    interpolate(24, 24, "p24_uni.txt");
```


```
    // - the actual function for Runge.
    Matrix avals = Linspace(-4, 4, 101);
    Matrix bvals = Linspace(-4, 4, 201);

    Matrix runge(101, 201);
    for(int i = 0; i < avals.Size(); i++) {
        for(int j = 0; j < bvals.Size(); j++) {
            // get all of the actual f values for comparison to our interpolated points
            runge(i, j) = f(avals(i), bvals(j));
        }
    }

    runge.Write("Runge.txt");
    return 0;
}
```

test_Newton.cpp



```
/* Erik Gabrielsen
   MATH 3316
   Program 3 */
   #include <stdlib.h>
   #include <stdio.h>
   #include <iostream>
   #include <math.h>
   #include "matrix.hpp"

   using namespace std;


   // function prototypes
   Matrix Newton_coefficients(Matrix& x, Matrix& y);
   double Newton_evaluate(Matrix& x, Matrix& c, double z);
   double f(double x) {
     return cosh(2.0*x*x);
   }


   // This routine tests the function newton_interpolant.cpp
  int main(int argc, char* argv[]) {

    vector<size_t> nvals = {10, 20};
    for (size_t k=0; k<nvals.size(); k++) {
      int n = nvals[k];
      cout << endl << "interpolants and errors using " << n << " nodes:\n";

      Matrix x = Linspace(0.0, 1.0, n+1, 1);
      Matrix y(n+1);
      for(int i = 0; i <= n; i++) {
          y(i) = f(x(i));
      }
      double dx = 1.0/n;
      Matrix z = Linspace(dx/2.0, 1.0-dx/2.0, n, 1); // evaluation points

      Matrix p(n); // results of interpolation
      Matrix c(n+1); // coefficients
      c = Newton_coefficients(x, y); // only need to calculate the coefficients once
      for(int i = 0; i < n; i++) {
          // calculate Newton interpolant at each evaluation point
          p(i) = Newton_evaluate(x, c, z(i));
      }

      cout << endl;
      cout << "    z        f(z)                p(z)              error" << endl;
      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)));
      }

    }
    return 0;
```

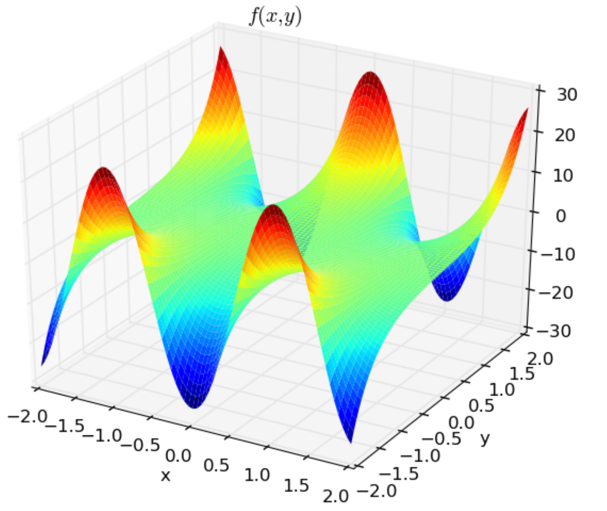
Runge2D.py

```
ax.set_ylabel('y')
title('$f(x,y) - uniform_p24(x,y)$')runge = 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')
a = loadtxt('avals.txt')
b = loadtxt('bvals.txt')

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, runge, rstride=1, cstride=1, linewidth=0,
cmap=cm.jet)
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$f(x,y)$')
savefig('runge.png')

fig = figure()
ax = fig.add_subplot(111, projection='3d')
X, Y = meshgrid(b, a)
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('$Uniform p6(x,y)$')
savefig('p6_uni.png')

fig = figure()
ax = fig.add_subplot(111, projection='3d')
X, Y = meshgrid(b, a)
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('$Uniform p24(x,y)$')
savefig('p24_uni.png')

fig = figure()
ax = fig.add_subplot(111, projection='3d')
X, Y = meshgrid(b, a)
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('$Chebyshev p6(x,y)$')
savefig('p6_Cheb.png')

fig = figure()
ax = fig.add_subplot(111, projection='3d')
X, Y = meshgrid(b, a)
surf = ax.plot_surface(X, Y, p24_Cheb, rstride=1, cstride=1, linewidth=0,
cmap=cm.jet)
```

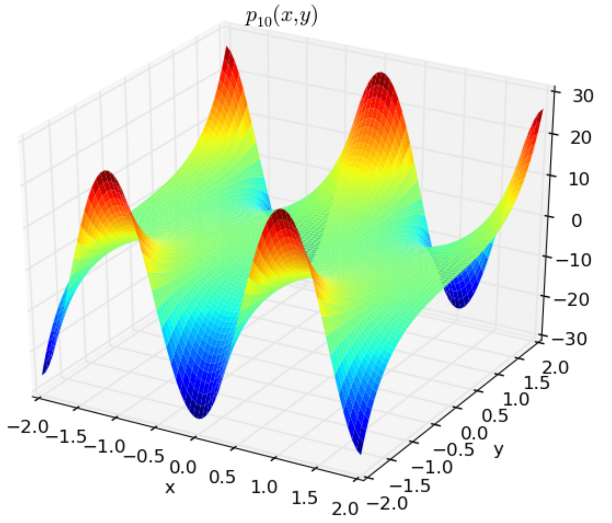
```
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$Chebyshev p24(x,y)$')
savefig('p24_Cheb.png')

# Errors
err_6_uni = abs(runge - p6_uni)
fig = figure()
ax = fig.add_subplot(111, projection='3d')
X, Y = meshgrid(b, a)
surf = ax.plot_surface(X, Y, err_6_uni, rstride=1, cstride=1, linewidth=0,
cmap=cm.jet)
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$f(x,y) - uniform_p6(x, y)$')
savefig('err_6_uni.png')

err_24_uni = abs(runge - p24_uni)
fig = figure()
ax = fig.add_subplot(111, projection='3d')
X, Y = meshgrid(b, a)
surf = ax.plot_surface(X, Y, err_24_uni, rstride=1, cstride=1, linewidth=0,
cmap=cm.jet)
ax.set_xlabel('x')
savefig('err_24_uni.png')

err_6_Cheb = abs(runge - p6_Cheb)
fig = figure()
ax = fig.add_subplot(111, projection='3d')
X, Y = meshgrid(b, a)
surf = ax.plot_surface(X, Y, err_6_Cheb, rstride=1, cstride=1, linewidth=0,
cmap=cm.jet)
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$f(x,y) - Chebyshev_p6(x,y)$')
savefig('err_6_Cheb.png')

err_24_Cheb = abs(runge - p24_Cheb)
fig = figure()
ax = fig.add_subplot(111, projection='3d')
X, Y = meshgrid(b, a)
surf = ax.plot_surface(X, Y, err_24_Cheb, rstride=1, cstride=1, linewidth=0,
cmap=cm.jet)
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$f(x,y) - Chebyshev_p24(x,y)$')
savefig('err_24_Cheb.png')
```

Appendix II – Graphs

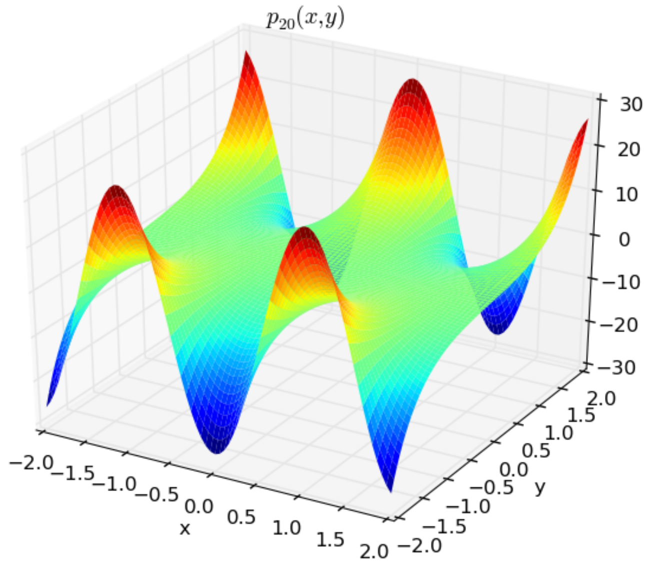
Lagrange2D – f(x, y)



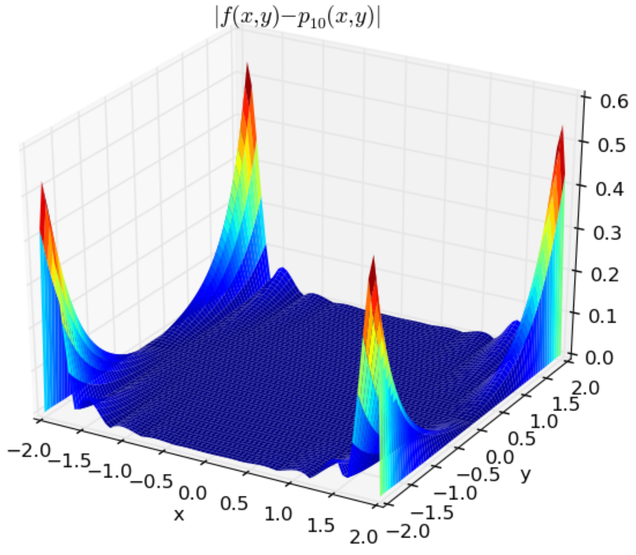
Lagrange2D – p10(x, y)



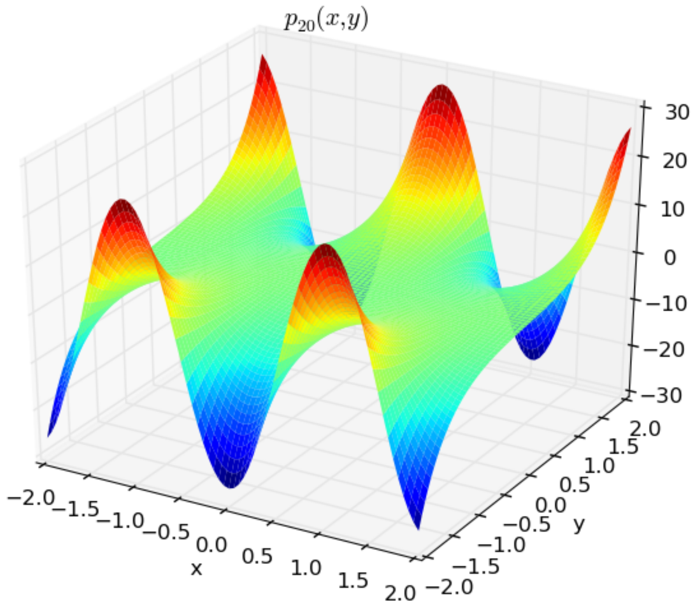
Lagrange 2D – p20(x, y)



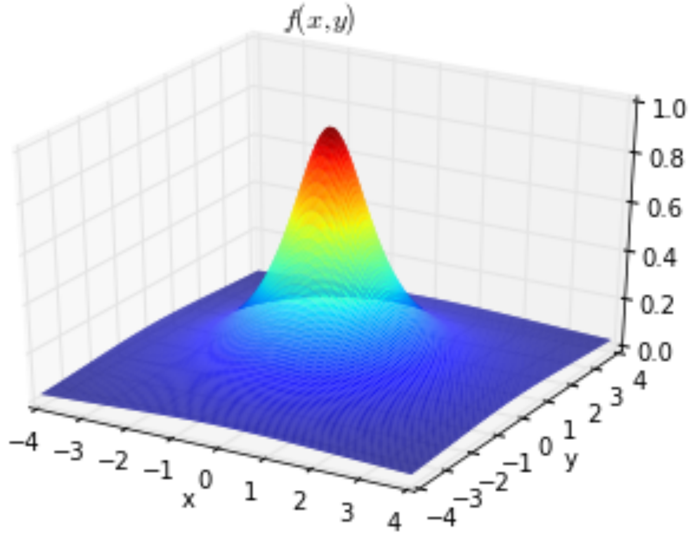
Lagrange 2D – f(x, y) – p10(x, y)



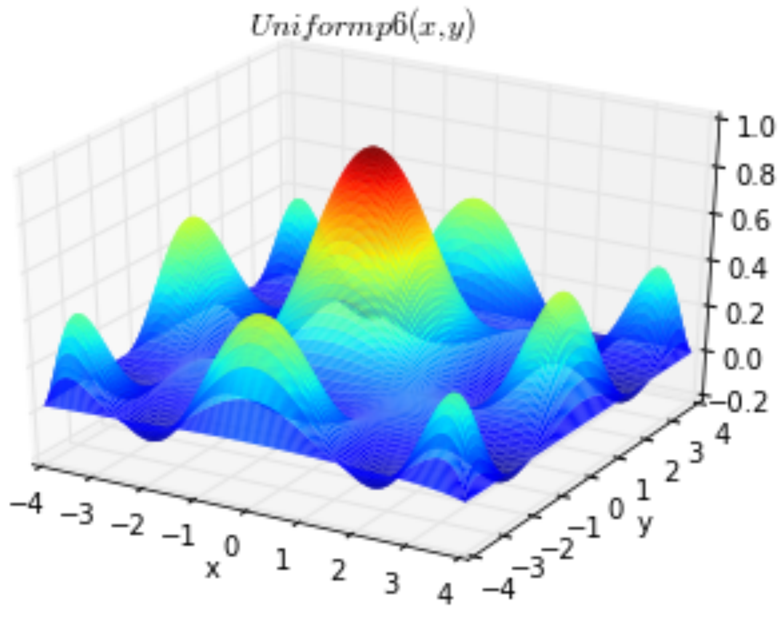
Lagrange 2D – f(x, y) – p20(x, y)



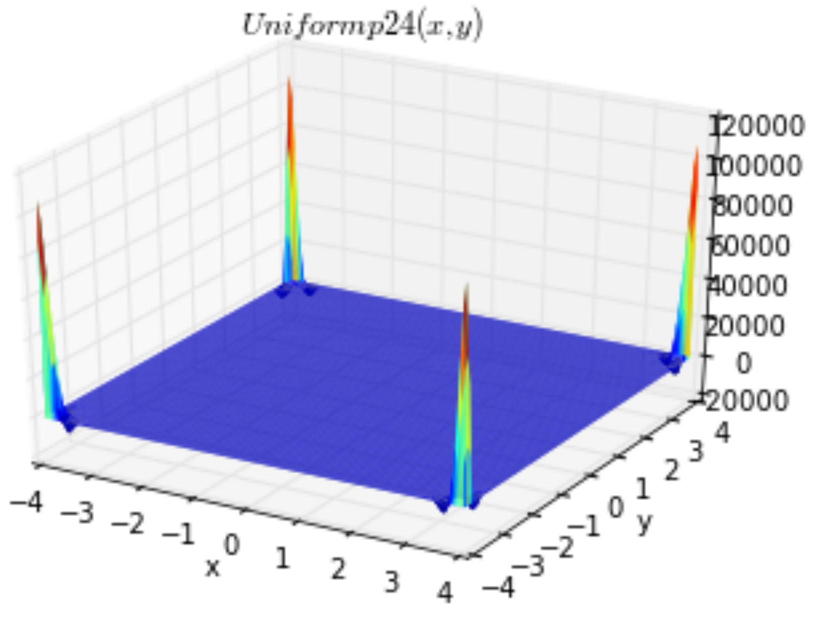
Runge – f(x,y)



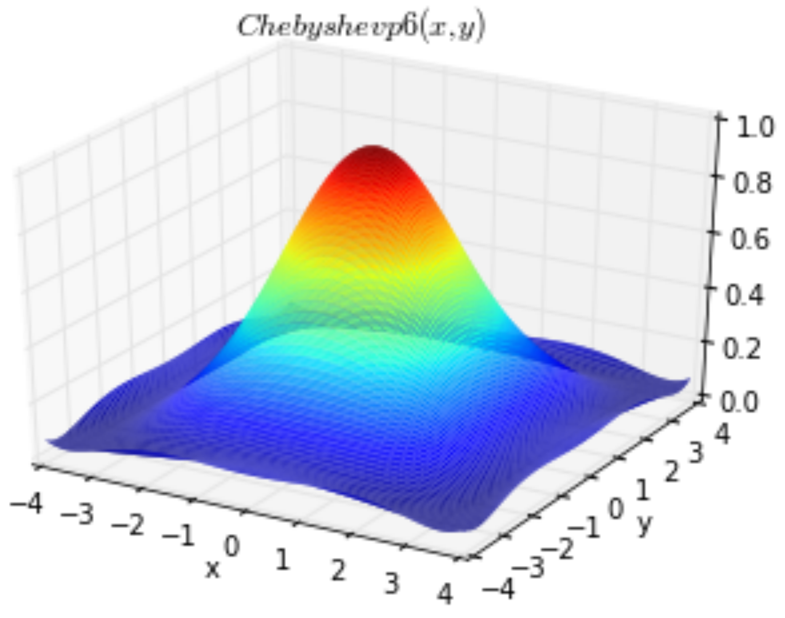
Runge – uniform p6(x, y)



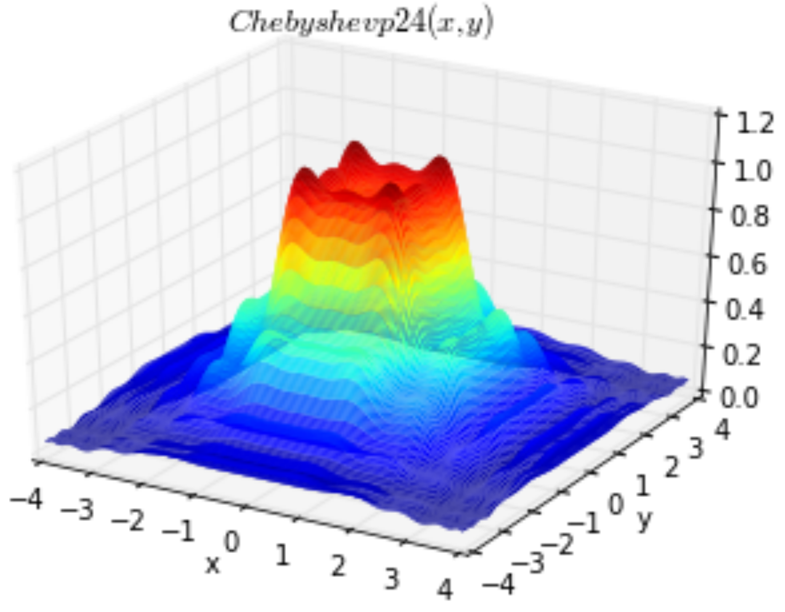
Runge – uniform p24(x, y)



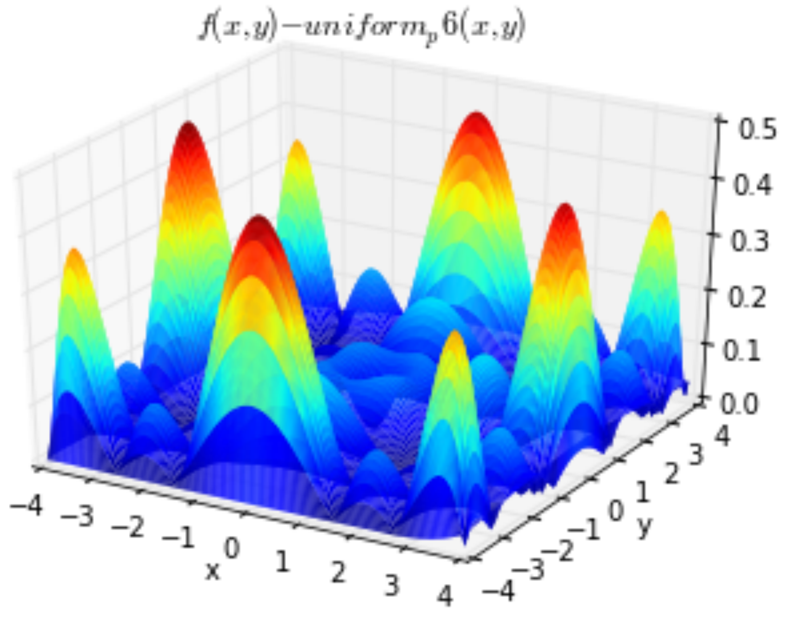
Runge – Chebyshev p6(x, y)



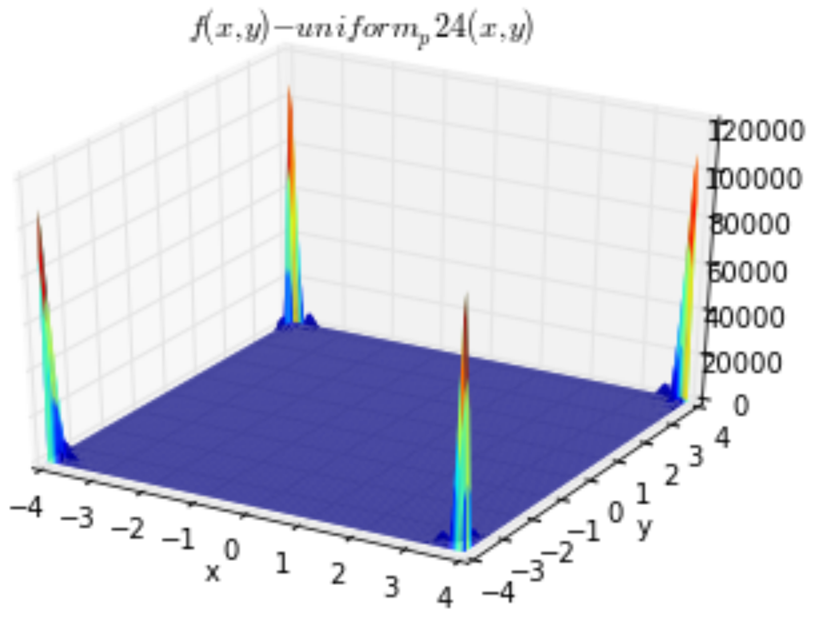
Runge – Chebyshev p24(x, y)



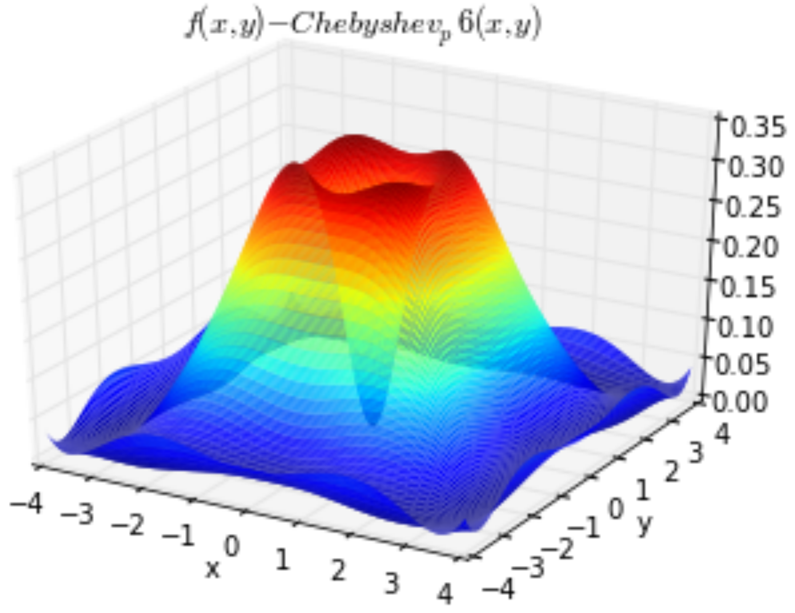
Runge – f(x, y) – uniform p6(x, y)



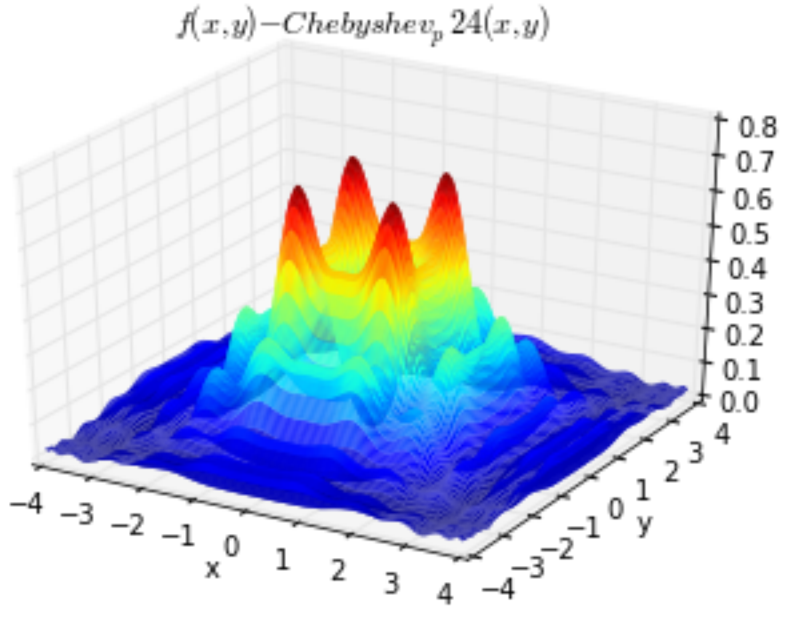
Runge – f(x, y) – uniform p24(x, y)



Runge – f(x, y) – Cheb p6(x, y)



Runge – f(x, y) – Cheb p24(x, y)


$f(x,y) - Chebyshev_p\, 24(x,y)$
x
y