Project 3 - Polynomial Interpolation

Conrad Appel - 11/04/2014 - MATH 3316

In part one of this project, I'll be implementing Newton's method of interpolating functions with a polynomial and comparing its performance to that of a given implementation of Lagrange's method of interpolation, tested using different sets of nodes and evaluation points. In part two, I will reimplement Lagrange's method for finding an interpolating polynomial, but adapted for two dimensions. In the final section, I will look at the difference between using evenly-spaced nodes and using what are called Chebyshev nodes, and the difference each makes on the error in the interpolating polynomial.

Part 1: Newton Interpolation

In this first section, I had to implement two functions in order to be able to compute the Newton interpolating polynomial:

```
int newton_coeffs(Mat& x, Mat& y, Mat& c);
double newton_eval(Mat& x, Mat&c, double z);
```

With the first function, `newton_coeffs`, I was able to calculate the coefficients by using the x- and actual y-values along with the algorithm summarized on page 166 of our textbook. Because the coefficients will be the same as long as the nodes and function don't change, I only needed to calculate them once in total to be used in evaluating all the interpolants.

With coefficients in hand, I was able to calculate the value of the interpolating polynomial at any point using my `newton_eval` function, passing in the x-values of the nodes, the calculated coefficients, and the x-value at which I want to find the value of the polynomial. With those in hand, I used the formula that defines the polynomial at a point:

```
p(z) = sum[from i=0 to n](c(i)*product[from j=0 to i-1](z - x(j)))
```

In order to make sure the evaluation was correct, I needed to create a clone of the provided `test_lagrange.cpp` file (called `test_newton.cpp`) that would test the Newton interpolation on function `f(x) = e^(-x^2)` with 6 and 12 nodes and evaluated from -2 to 2 and from -1 to 1, respectively.

Output of `test_newton.o`

```
interpolant/error w/ 6 nodes:
   z          f(z)                  p(z)                  error
  -1.600   0.0773047404433  -0.0247167527608   0.102021
  -0.800   0.5272924240430   0.5819135611732   0.0546211
   0.000   1.0000000000000   0.9527603226136   0.0472397
   0.800   0.5272924240430   0.5819135611732   0.0546211
   1.600   0.0773047404433  -0.0247167527609   0.102021

interpolant/error w/ 12 nodes:
   z          f(z)                  p(z)                  error
  -0.818   0.5120046931872   0.5119191568060   8.55364e-05
  -0.655   0.6515313576784   0.6514572253658   7.41323e-05
  -0.491   0.7858481209741   0.7858817953300   3.36744e-05
  -0.327   0.8984291423598   0.8984899459678   6.08036e-05
  -0.164   0.9735784620689   0.9735693052746   9.15679e-06
   0.000   1.0000000000000   0.9999431530739   5.68469e-05
   0.164   0.9735784620689   0.9735693052746   9.15679e-06
   0.327   0.8984291423598   0.8984899459678   6.08036e-05
   0.491   0.7858481209741   0.7858817953300   3.36744e-05
   0.655   0.6515313576784   0.6514572253658   7.41323e-05
   0.818   0.5120046931872   0.5119191568060   8.55364e-05
```

The output of this program is the result of two trials, once with 6 nodes, and once with 12 nodes. In both trials, the nodes and evaluation points are equally spaced. In the first, 6 nodes x were spread out from -2 to 2 and evaluation points z were spread between the nodes. In this trial at the evaluation points, there was an average error of about .072. In the second trial, 12 nodes x were spread also from -2 to 2, but evaluation points were equally-spaced from -1 to 1 (due to a mistyping in the project description, but the range for these points shouldn't be relevant as they still fall within the range of points used to do the interpolation). The average error for this trial was about .0000648. Compared to the trial with 6 nodes, the trial with 12 nodes was able to interpolate points much nearer to the actual value, which is what we'd expect, as more nodes means the interpolated polynomial should be correct at more points and therefore should lie closer to the actual function.

Next, I'm going to compare the execution time of my implemented Newton's method with the given Lagrange method, using differing numbers of nodes (5, 10, 20 40) and differing numbers of evaluation points (500, 5000, 50000, 500000). The Lagrange will be run as given, and the Newton method will have its coefficients calculated for each n, then evaluated at each evaluation point m. Results are discarded as the primary purpose of this section is to measure general performance.

Example output of `compare.o`

```
With n = 5 and m = 500:
   Lagrange time: 0.149115 milliseconds
   Newton time:   0.061156 milliseconds
```

```
With n = 5 and m = 5000:
   Lagrange time: 1.50991 milliseconds
   Newton time:   0.639694 milliseconds
With n = 5 and m = 50000:
   Lagrange time: 17.3907 milliseconds
   Newton time:   7.36335 milliseconds
With n = 5 and m = 500000:
   Lagrange time: 164.849 milliseconds
   Newton time:   62.3255 milliseconds
With n = 10 and m = 500:
   Lagrange time: 0.442327 milliseconds
   Newton time:   0.172938 milliseconds
With n = 10 and m = 5000:
   Lagrange time: 4.3203 milliseconds
   Newton time:   1.75495 milliseconds
With n = 10 and m = 50000:
   Lagrange time: 46.8562 milliseconds
   Newton time:   17.4201 milliseconds
With n = 10 and m = 500000:
   Lagrange time: 443.251 milliseconds
   Newton time:   174.57 milliseconds
With n = 20 and m = 500:
   Lagrange time: 1.42998 milliseconds
   Newton time:   0.65386 milliseconds
With n = 20 and m = 5000:
   Lagrange time: 16.8275 milliseconds
   Newton time:   6.42598 milliseconds
With n = 20 and m = 50000:
   Lagrange time: 147.672 milliseconds
   Newton time:   63.747 milliseconds
With n = 20 and m = 500000:
   Lagrange time: 1431.61 milliseconds
   Newton time:   642.207 milliseconds
With n = 40 and m = 500:
   Lagrange time: 5.34851 milliseconds
   Newton time:   2.29733 milliseconds
With n = 40 and m = 5000:
   Lagrange time: 59.0664 milliseconds
   Newton time:   23.2833 milliseconds
With n = 40 and m = 50000:
   Lagrange time: 545.842 milliseconds
   Newton time:   232.182 milliseconds
With n = 40 and m = 500000:
   Lagrange time: 5416.93 milliseconds
   Newton time:   2332.03 milliseconds
```

With each increase of n, the times for both the Lagrange and Newton methods increased

dramatically, almost seemingly exponentially. However, Newton's method started out at about a third of the time of Lagrange and continued to remain at about a third of Lagrange's time throughout all of the trials. I believe this time difference is due to the time complexity needed to compute each method. It looks as if Newton's method, especially after needing to calculate its coefficients only once to reuse in all the point calculations, is only of O(nlogn) time complexity, whereas Lagrange's method looks as if it falls within O(n^2) time complexity. A time complexity of O(nlogn) is much less than O(n^2), and as such would provide a good explanation as to why Newton's method consistently runs in much less time than Lagrange's method. Additionally, evaluating the interpolant at more points m seemed to linearly increase the time needed for evaluation, multiplying the evaluation time by about ten whenever the number of evaluation points was increased by ten times.

Part 2: Multi-dimensional interpolation

In this second section, I will adapt the Lagrange method to be used in a 2D setting. To do this the Lagrange bases are calculated for each axis and then multiplied together to form the two-dimensional basis, which can be used in nearly the same way as the basis is used in the one-dimensional variant. However, the 2D version should be adapted as such, with m y-values and n x-values and with l(x, y) being the product of the two one-dimensional bases at points x and y:

```
p(a, b) = sum[from i=0 to m]( sum[from j=0 to n]( f(x(i), y(j)) * l[i, j](a, b) ))
```

I was able to reuse the Lagrange basis method provided in `lagrange.cpp` to form the two-dimensional basis, as it uses the same technique at the heart of the method.

Using the provided `test_lagrange2D.cpp` and `plot_lagrange2D.py` files, I tested my `lagrange2D()` function, and used the output to generate surface plots of the results (Figures 1-5 at the end of the document). The output of `plot_lagrange2D.py` is as follows:

```
p6 success!  ||e6|| =  0.0124815891496  is below tolerance of 0.013
p15 success!  ||e15|| =  3.84877534218e-08  is below tolerance of 4e-8
```

The errors in both of the polynomial interpolations seemed to be at an acceptable value for what is expected of them. To the naked eye, it seems as if the six- and fifteen-node interpolated functions are nearly exactly the same as the actual values of f, which is what I would have hoped for. The error for the 15-node interpolation is also much smaller within the range of the nodes, which is also expected behavior.

While the one-dimensional Lagrange interpolation method from part one had an upper-bound of O(n^2), because this two-dimensional variant adds another nested loop for the y-values, I believe the time complexity of this algorithm would be bumped up to O(n^3).

Part 3: The importance of nodes

In the third section of this project, I will demonstrate how using evenly-spaced nodes can end up worse than using a set of more wisely-chosen nodes. To show this phenomenon, I'll use the Runge function:

```
f(x, y) = 1 / (1 + x^2 + y^2)
```

When creating a polynomial interpolation of this function (plotted in figure 6), it seems that creating a higher-order polynomial with more equally-spaced nodes actually provides a worse approximation of the function than with less nodes, which is antithetical to what I witnessed in the first two sections of this project.

The function and the two equally-spaced node interpolations are all evaluated at 20301 points. The actual function values range from 0 to 1 at any point (figure 6). The eight-node equally-spaced node interpolation's values range from about -1.27 to about 2.43 within the limits of the nodes (figure 7). Normally, I would expect the interpolation with more nodes to be closer to the actual values of the function, but in this case, the 16-node equally-spaced interpolation (figure 8) contains values ranging all the from around -190 to 1998, considerably worse than the eight-node interpolation.

Using a different method to determine the nodes seems to be the solution to this problem. As suggested in the project description, I implemented what are called Chebyshev nodes, which were created to resolve this issue and allow a higher number of nodes create a better approximation. To find the Chebyshev nodes, I used the following formula, where L is the range of the interval of the nodes and m is the number of nodes - 1:

```
x(i) = L * cos(pi*(2i + 1) / (2m + 2))
```

Using these nodes, I was able to achieve much better results. At the same 20301 points, I evaluated the eight- and sixteen-node polynomial interpolants, but this time using the Chebyshev nodes. This time, with the eight-node interpolation, the values ranged from -.05 to 1, and with the sixteen-node interpolation, the values ranged from around .018 to 1, both of which are much closer to the actual function. The surface plot of the sixteen-node interpolation does look both much more similar to the actual graph of f, as well as being a much smoother and accurate approximation than the eight-value interpolation.

Overall, in terms of approximation quality of this Runge function, I rank the following methods from best to worst:

1. 16 Chebyshev nodes
2. 8 Chebyshev nodes
3. 8 evenly-spaced nodes

4. 16 evenly-spaced nodes

Appendix A: Code

Makefile

```
# Conrad Appel
# MATH 3316
# Nov 4 2014

all: test_newton compare test_lagrange2d runge_regular runge_chebyshev

test_newton:
	g++ -std=c++11 test_newton.cpp mat.cpp -o test_newton.o

compare:
	g++ -std=c++11 compare.cpp mat.cpp -o compare.o

test_lagrange2d:
	g++ -std=c++11 test_lagrange2D.cpp lagrange2D.cpp mat.cpp -o lagrange2D.o

runge_regular:
	g++ -std=c++11 runge_regular.cpp mat.cpp -o runge_regular.o

runge_chebyshev:
	g++ -std=c++11 runge_chebyshev.cpp mat.cpp -o runge_chebyshev.o

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

newton_interp.cpp

```cpp
// Conrad Appel
// MATH 3316
// Nov 4 2014

#include <iostream>

#include "mat.h"

int newton_coeffs(Mat&, Mat&, Mat&);
double newton_eval(Mat&, Mat&, double);

int newton_coeffs(Mat& x, Mat& y, Mat& c) {
    unsigned int n = x.Cols();
    // algorithm described in book (summarized on page 166)
    for(unsigned int i = 0; i < n; i++) {
        c(i) = y(i);
    }
    for(unsigned int i = 1; i < n; i++) {
        for(unsigned int j = n-1; j >= i; j--) {
            c(j) = (c(j) - c(j-1))/(x(j) - x(j-i));
        }
    }
    return 0;
}

// evaluates Newton interpolation polynomial with nodes x, coefficients c, at point z
double newton_eval(Mat& x, Mat& c, double z) {
    unsigned int n = x.Cols();
    double sum = 0;
    for(int i = 0; i < n; i++) {
        double product = 1;
        for(unsigned int j = 0; j < i; j++) {
            product *= (z - x(j));
        }
        sum += c(i)*product;
    }
    return sum;
}
```

test_newton.cpp

```cpp
// Conrad Appel
// MATH 3316
// Nov 4 2014

#include <cmath>
#include <iostream>
```

```
#include "newton_interp.cpp"
#include "mat.h"

// f(x) = e^(-x^2)
double f(const double x) {
    return exp(-x*x);
}

// used to reuse code for 8 and 16 nodes
void testWithNNodes(int n, double range) {
    n -= 1;
    Mat x = Linspace(-2.0, 2.0, n+1);
    Mat y(n+1);
    for(int i = 0; i < n+1; i++) {
        y(i) = f(x(i));
    }
    double dx = 4.0/n;
    Mat z = Linspace(-range+dx/2.0, range-dx/2.0, n); // evaluation points

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

    std::cout << "interpolant/error w/ " << n+1 << " nodes: " << std::endl;
    std::cout << "    z       f(z)              p(z)             error" << std::endl;
    for(int i = 0; i < n; i++) {
        printf("   %6.3f   %16.13f   %16.13f   %g\n", z(i), f(z(i)), p(i), std::abs(f(z(i)) - p(i)));
    }

    std::cout << std::endl << std::endl;
}

int main()
{
    testWithNNodes(6, 2.0);
    testWithNNodes(12, 1.0);
    return 0;
}
```

compare.cpp

```
// Conrad Appel
```

```
// MATH 3316
// Nov 4 2014

#include "mat.h"
#include "newton_interp.cpp"
#include "lagrange.cpp"

#include <cmath>
#include <chrono>

// used to easily run the Lagrange method with differing values of n and m
void testLagrange(unsigned int n, Mat& x, Mat& y, Mat& z) {
   Mat p(z.Size());
   for(unsigned int i = 0; i < z.Size(); i++) {
      lagrange(x, y, z(i));
   }
}

// used to easily run Newton's method with differing values of n and m
void testNewton(unsigned int n, Mat& x, Mat& y, Mat& z) {
   Mat p(z.Size());
   Mat c(n+1);
   newton_coeffs(x, y, c);
   for(int i = 0; i < z.Size(); i++) {
      p(i) = newton_eval(x, c, z(i));
   }
}

int main()
{
   unsigned int n_possible[4] = {5, 10, 20, 40}; // possible number of nodes
   unsigned int m_possible[4] = {500, 5000, 50000, 500000}; // possible number of
evaluation points
   for(unsigned int i = 0; i < 4; i++) {
      for(unsigned int j = 0; j < 4; j++) {
         unsigned int n = n_possible[i];
         unsigned int m = m_possible[j];

         Mat x = Linspace(-1, 1, n+1);
         Mat y(n+1);
         for(unsigned int k = 0; k < n+1; k++) {
            y(k) = std::sinh(std::pow(x(k), 3)/2.0); // evaluate the function at all nodes
         }
         Mat z = Linspace(-1, 1, m+1);

         // time the evaluation time of running the Lagrange method at selected n and
m
         auto start = std::chrono::system_clock::now();
         testLagrange(n, x, y, z);
```

```
            auto stop = std::chrono::system_clock::now();
            double LGTime =
std::chrono::duration_cast<std::chrono::nanoseconds>(stop-start).count();

            // time the evaluation time of running the Newton method at selected n and m
            start = std::chrono::system_clock::now();
            testNewton(n, x, y, z);
            stop = std::chrono::system_clock::now();
            double NewTime =
std::chrono::duration_cast<std::chrono::nanoseconds>(stop-start).count();

            std::cout << "With n = " << n << " and m = " << m << ": " << std::endl;
            std::cout << '\t' << "Lagrange time: " << LGTime*1e-6 << " milliseconds" <<
std::endl;
            std::cout << '\t' << "Newton time:   " << NewTime*1e-6 << " milliseconds" <<
std::endl;
            std::cout << std::endl;
        }
    }

    return 0;
}
```

lagrange2D.cpp

```
// Conrad Appel
// MATH 3316
// Nov 4 2014

#include "mat.h"
#include "lagrange.cpp"

// evaluates the Lagrange interpolated polynomial at specified 2D point (x, y)
double lagrange2D(Mat& x, Mat& y, Mat& f, double a, double b) {
    double sum = 0;
    // implements equation (2) from the project description
    for(int i = 0; i < x.Size(); i++) {
        for(int j = 0; j < y.Size(); j++) {
            sum += f(i, j) * lagrange_basis(x, i, a) * lagrange_basis(y, j, b);
        }
    }
    return sum;
}
```

runge_regular.cpp

```cpp
// Conrad Appel
// MATH 3316
// Nov 4 2014

#include <iostream>
#include <cmath>

#include "mat.h"
#include "lagrange2D.cpp"

// 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 computeWithNodes(unsigned int n, unsigned int m, std::string filename) {
   Mat x = Linspace(-6, 6, m+1);
   Mat y = Linspace(-6, 6, n+1);

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

   Mat avals = Linspace(-6, 6, 101);
   Mat bvals = Linspace(-6, 6, 201);

   Mat p(101, 201);
   for(unsigned int i = 0; i < avals.Size(); i++) {
      for(unsigned 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() {
   computeWithNodes(8, 8, "./p8_reg.txt");
   computeWithNodes(16, 16, "./p16_reg.txt");

   Mat avals = Linspace(-6, 6, 101);
```

```
    Mat bvals = Linspace(-6, 6, 201);

    Mat runge(101, 201);
    for(unsigned int i = 0; i < avals.Size(); i++) {
        for(unsigned 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;
}
```

runge_chebyshev.cpp

```
// Conrad Appel
// MATH 3316
// Nov 4 2014

#include <iostream>
#include <cmath>

#include "mat.h"
#include "lagrange2D.cpp"

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

// this method uses Chebyshev nodes in order to remedy the issues
// associated with equispaced interpolation points
// x(i) = L*cos(pi*(2i + 1)/(2m+2))
double chebNode(double n, double i, double m) {
    return n*std::cos((2*i + 1) * PI / (2*m + 2));
}

// used to reuse code for 8 and 16 nodes
void computeWithNodes(unsigned int n, unsigned int m, std::string filename) {
    Mat x(m+1);
    Mat y(n+1);
    for(unsigned int i = 0; i < x.Size(); i++) {
        // calculate all of the Chebyshev nodes
        x(i) = chebNode(6, i, m);
        y(i) = chebNode(6, i, m);
    }
```

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

    Mat avals = Linspace(-6, 6, 101);
    Mat bvals = Linspace(-6, 6, 201);

    Mat p(101, 201);
    for(unsigned int i = 0; i < avals.Size(); i++) {
        for(unsigned 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() {
    computeWithNodes(8, 8, "./p8_cheb.txt");
    computeWithNodes(16, 16, "./p16_cheb.txt");

    return 0;
}
```

runge2D.py

```
#!/usr/bin/env python

# Conrad Appel
# MATH3316
# Nov 4 2014

from pylab import *
from mpl_toolkits.mplot3d import Axes3D

f = loadtxt('./runge.txt')
p8_reg = loadtxt('./p8_reg.txt')
p16_reg = loadtxt('./p16_reg.txt')
p8_cheb = loadtxt('./p8_cheb.txt')
p16_cheb = loadtxt('./p16_cheb.txt')
```

```
p16_cheb = loadtxt('./p16_cheb.txt')

err_p8_reg = ndarray(shape=(101, 201))
err_p16_reg = ndarray(shape=(101, 201))
err_p8_cheb = ndarray(shape=(101, 201))
err_p16_cheb = ndarray(shape=(101, 201))

# error calculations
for i in range(101):
    for j in range(201):
        err_p8_reg[i, j] = abs(f[i, j] - p8_reg[i, j])
        err_p16_reg[i, j] = abs(f[i, j] - p16_reg[i, j])
        err_p8_cheb[i, j] = abs(f[i, j] - p8_cheb[i, j])
        err_p16_cheb[i, j] = abs(f[i, j] - p16_cheb[i, j])

avals = loadtxt('./avals.txt')
bvals = loadtxt('./bvals.txt')

X, Y = meshgrid(bvals, avals)

fig = figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, f)
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$f(x,y)$')
savefig('runge2d_f.png')

fig = figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, p8_reg)
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$p8reg(x,y)$')
savefig('runge_p8_reg.png')

fig = figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, p16_reg)
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$p16reg(x,y)$')
savefig('runge_p16_reg.png')

fig = figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, p8_cheb)
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$p8cheb(x,y)$')
```

```
savefig('runge_p8_cheb.png')

fig = figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, p16_cheb)
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$p16cheb(x,y)$')
savefig('runge_p16_cheb.png')

fig = figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, err_p8_reg)
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$err(p8reg(x,y))$')
savefig('runge_p8_reg_err.png')

fig = figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, err_p16_reg)
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$err(p16reg(x,y))$')
savefig('runge_p16_reg_err.png')

fig = figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, err_p8_cheb)
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$err(p8cheb(x,y))$')
savefig('runge_p8_cheb_err.png')

fig = figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, err_p16_cheb)
ax.set_xlabel('x')
ax.set_ylabel('y')
title('$err(p16cheb(x,y))$')
savefig('runge_p16_cheb_err.png')

show()
```

.

Appendix B: Plots

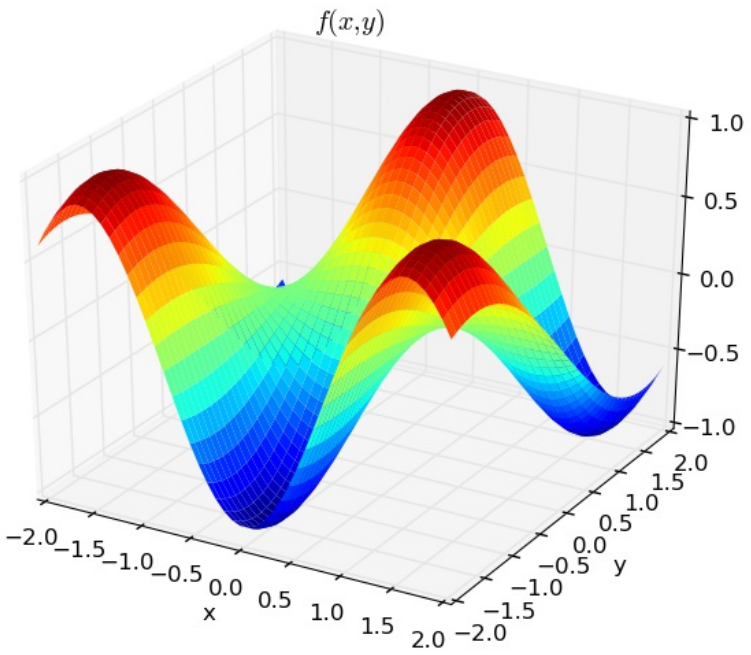

Figure 1: f(x, y), the true values of the two-dimensional function sin(x)*cos(2.0*y), whose values we hope to achieve in our interpolation.

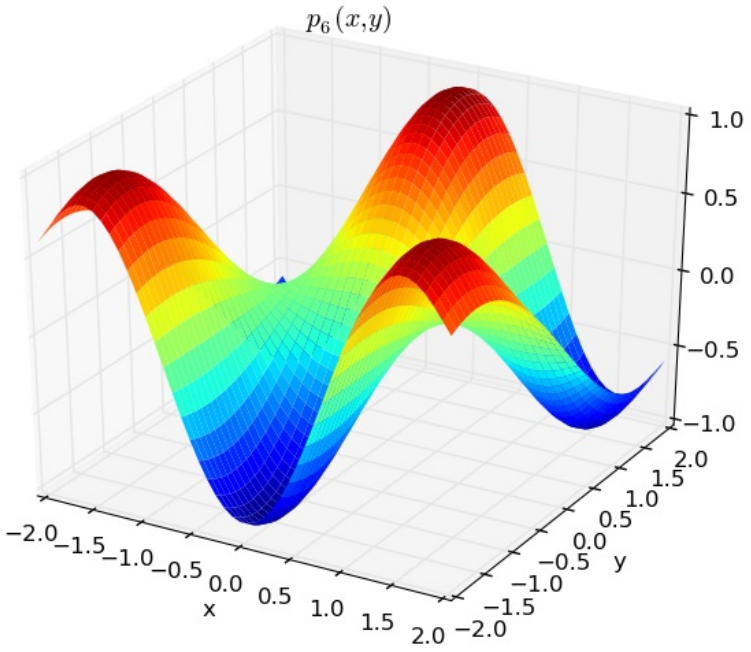

Figure 2: six-node two-dimensional Lagrange interpolation of sin(x)*cos(2.0*y)

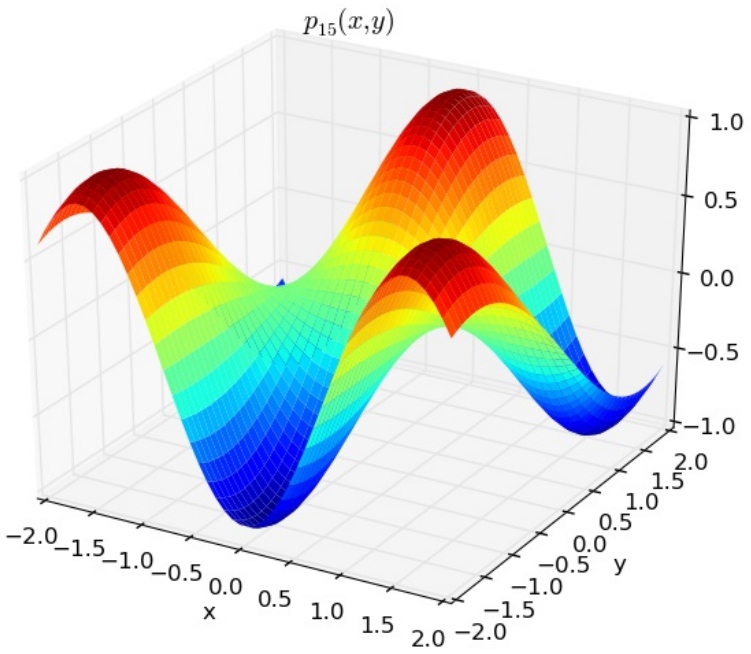

Figure 3: 15-node two-dimensional Lagrange interpolation of sin(x)*cos(2.0*y)

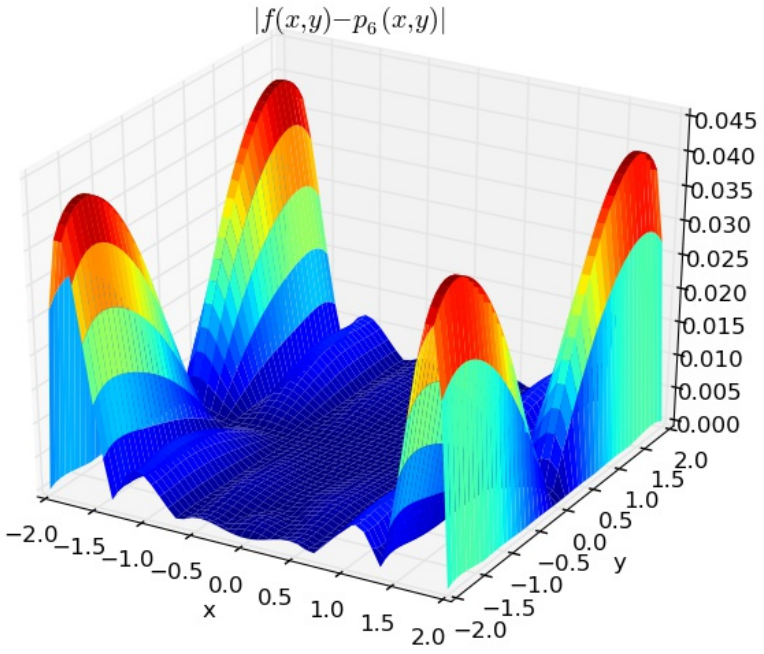

Figure 4: error between the six-node two-dimensional Lagrange interpolation and the actual values

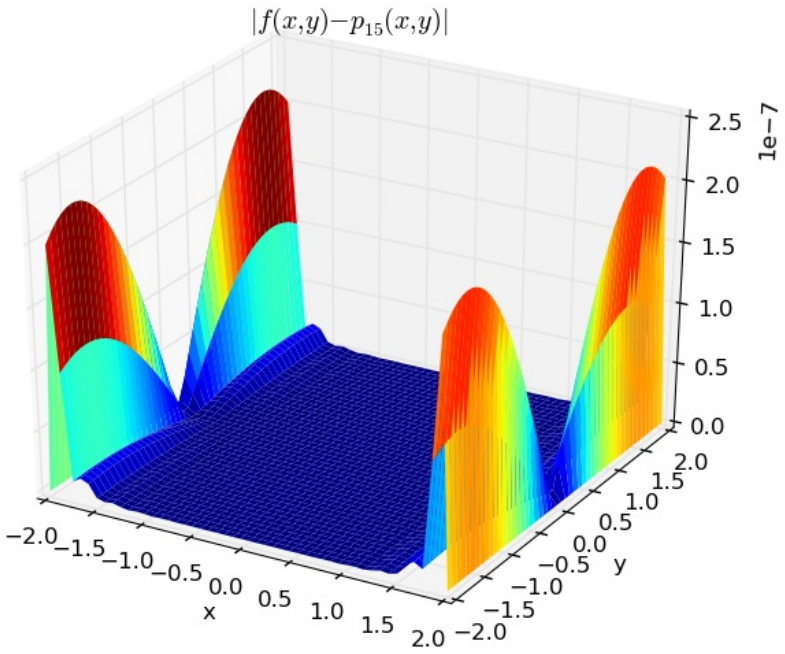

Figure 5: error between the 15-node two-dimensional Lagrange interpolation and the actual values

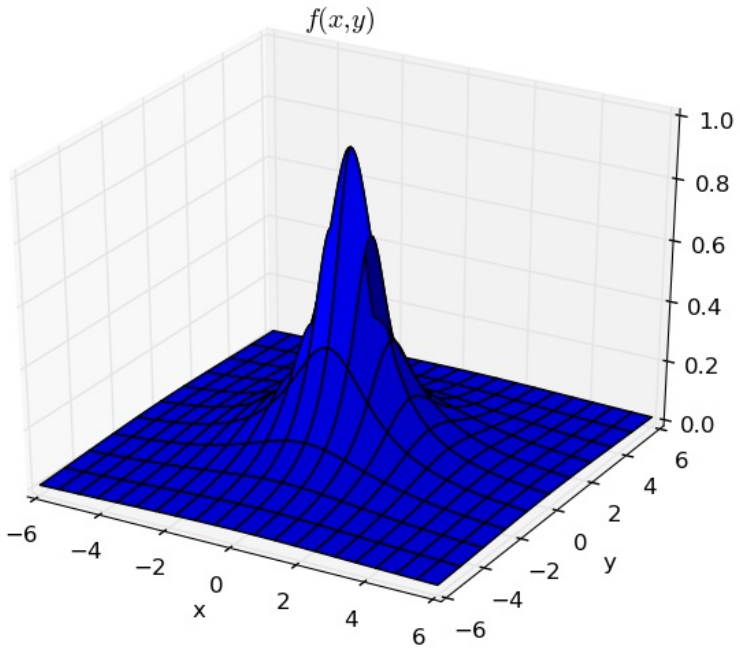

Figure 6: f(x, y), the true values of the two-dimensional Runge function and the values we strive to achieve in our polynomial interpolation.

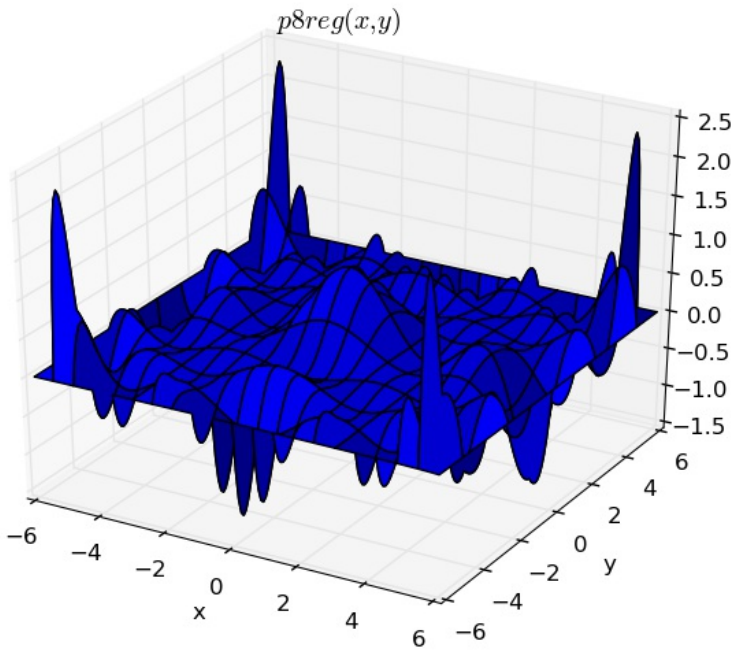

Figure 7: Runge function interpolated with 8 equally-spaced nodes.

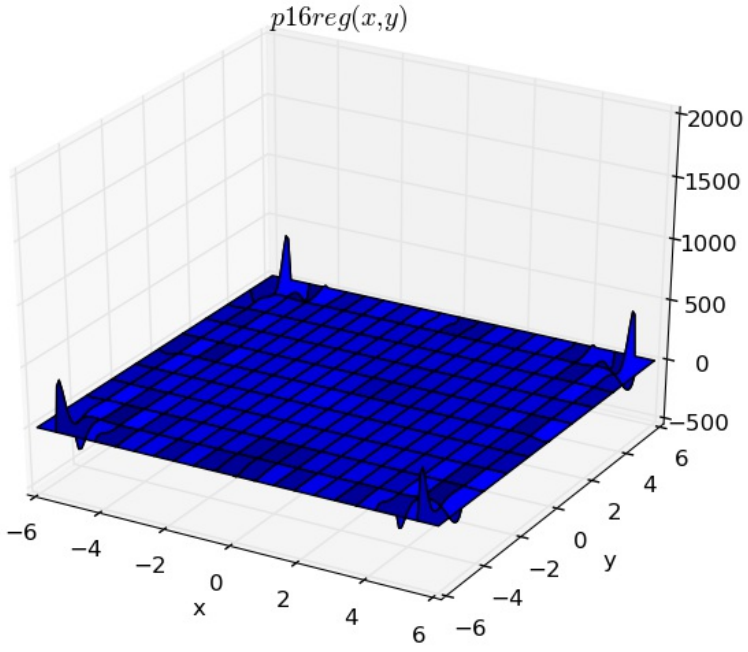

Figure 8: Runge function interpolated with 16 equally-spaced nodes. The true range of values is tough to see in the graph due to the huge jump in the outer corners, throwing off the scaling of the whole graph.

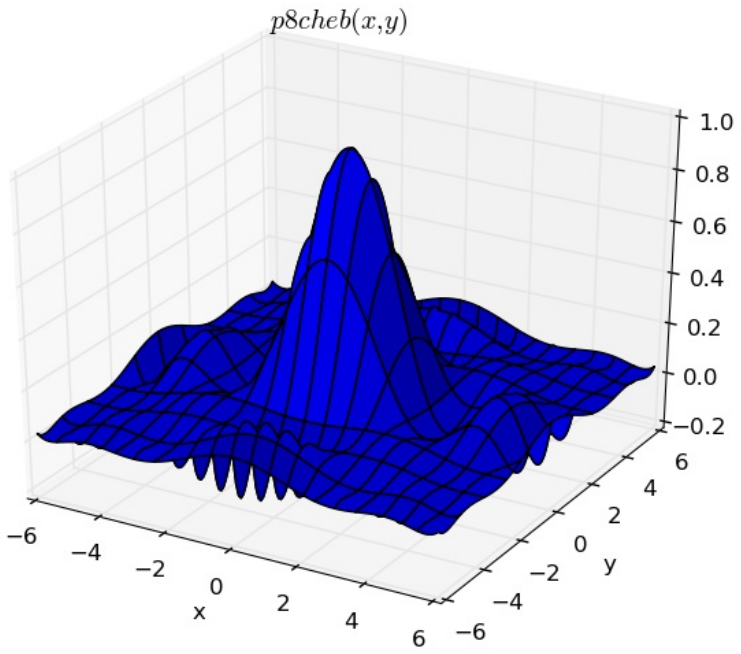

Figure 9: Runge function interpolated with 8 nodes determined with the Chebyshev node formula. This graph is more wave-like than the one with 16 nodes because the polynomial oscillates back and forth between the actual function values. In the 16-node interpolation, the interpolation is able to hug the actual function values more closely.

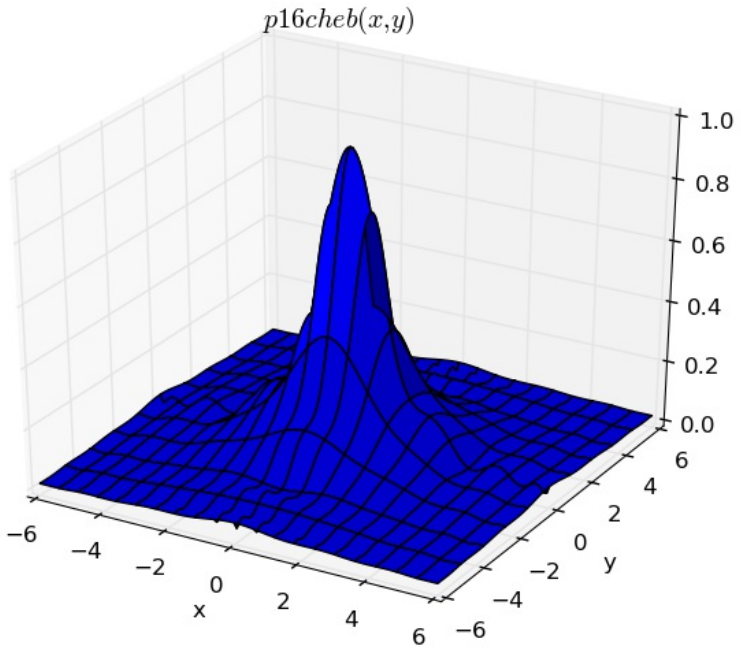

Figure 10: Runge function interpolated with 16 nodes determined with the Chebyshev node formula. Notice how much less wave-like this graph is than the previous one.

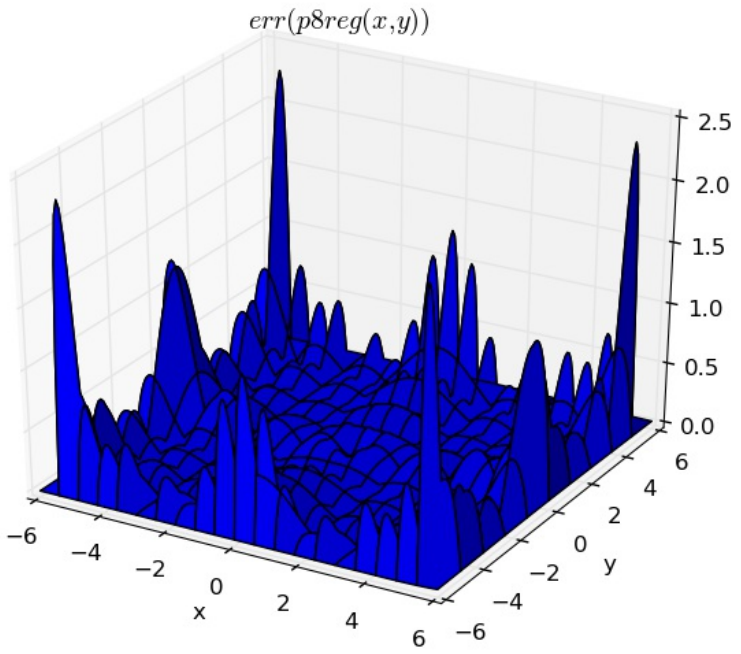

Figure 11: Error between the 8 equally-spaced node interpolation and the actual values of the Runge function.

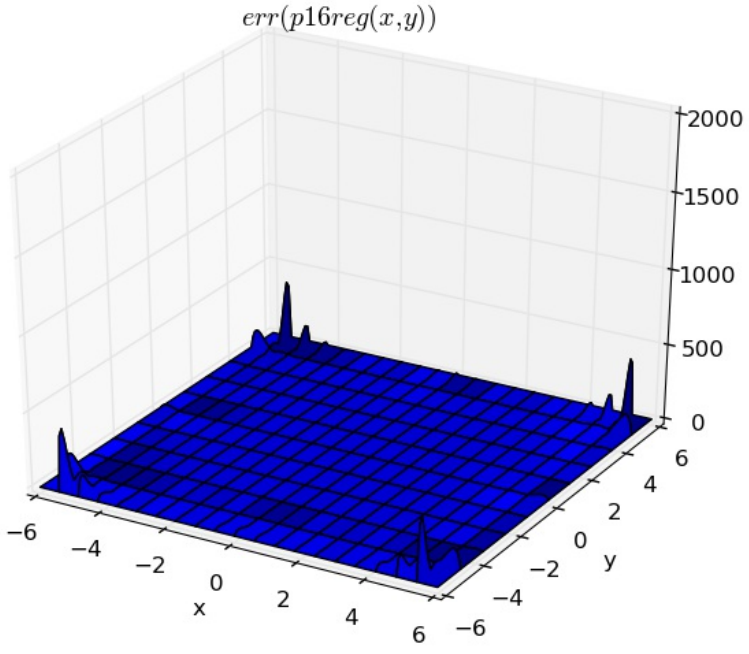

Figure 12: Error between the 16 equally-spaced node interpolation and the actual values of the Runge function.

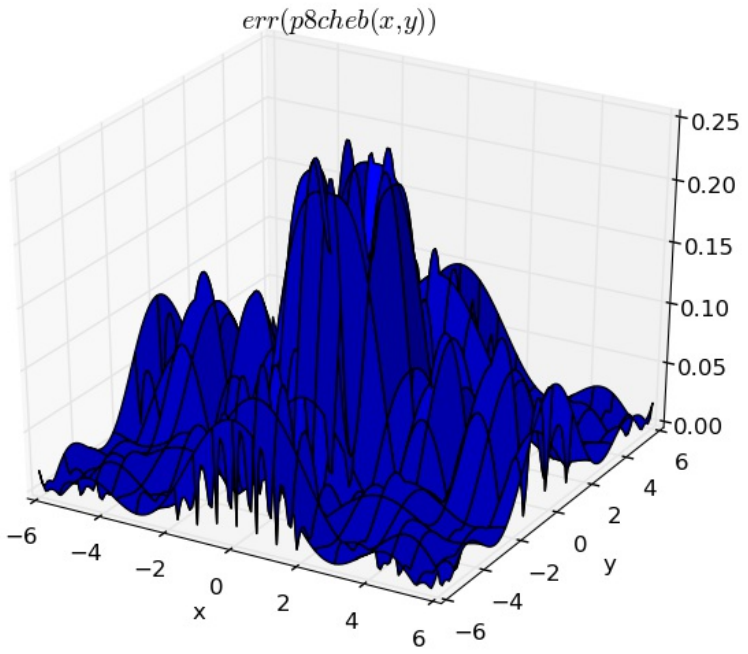

Figure 13: Error between the 8 Chebyshev node interpolation and the actual values of the Runge function.

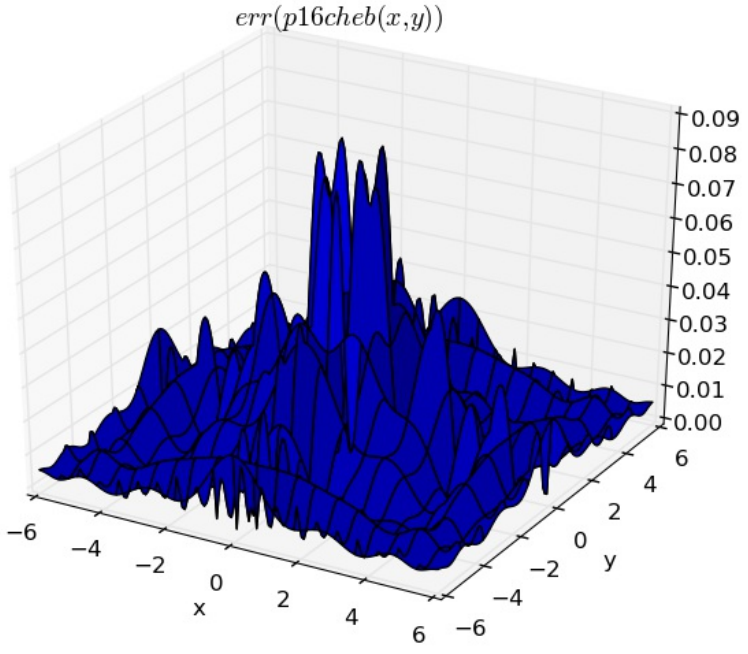

Figure 14: Error between the 16 Chebyshev node interpolation and the actual values of the Runge function.