Project 1

Van Tran

11/9/15

Math 3315

Part 1

In part 1 I created the newton function to evaluate the polynomial’s coefficients and a nested multiplication function that uses these coefficients in a nested multiplication function. I used the pseudocode given in the book on page 166. I also created a tester for the Newton interpolation which mimics the Lagrange tester given by the instructor. I created comparison program that compares how fast the newton and lagrange method is evaluated at different interpolating points and different points of evaluation. I used chrono and ctime to time the evaluation at each point.

No matter what m and n was, the evaluation of the newton and Lagrange method is almost exactly the same. This makes sense because the interpolating polynomial is unique, and therefore both the newton and lagrange solve for the same polynomial. The differences in error are extremely minor and are due to the difference in floating point error. The two methods have the same error on paper. The newton method was much faster than the Lagrange error and is apparent especially when the number of points evaluated grows (go ahead and compare first and last). The difference of time between the lagrange and newton evaluation of 100 points (m=100) using an interpolating polynomial of 10 (n = 10) points is .002105837, and the difference of time between the evaluation of 100000 points (m = 100000) using interpolating polynomial of 80 points (n = 80) is 36.613765.

This makes sense because as n grows larger, the difference in the complexity becomes more apparent. The lagrange function has a complexity of O(n^2). In the double Lagrange method, it has a for loop, looping over the data values n. Inside the loop it calls the lagrange basis method, which also loops over the data values n. n(n) is n^2. The newton evaluation of the coefficients has a complexity of O(n^2). There is a nested for loop in the Newton\_coefficients method which loops over n, n times. The newton evaluation given the coefficients is O(n), and can be seen that it contains a simple for loop iterating over the set n once. The reason why Newton is faster than Lagrange is because it only needs to evaluate the coefficients once and then evaluates it many times. So it only uses an n^2 algorithm once every time n changes and uses an n algorithm m times, whereas the Lagrange uses an n^2 algorithm m times. M\*(n^2) is obviously larger than (n^2) + M(n).

Part 2

In part 2 I created the 2d Lagrange interpolation function using the lagrange file given by the instructor. I used the lagrange basis method and called it twice within the lagrange2d method created. I then tested it using the test\_Lagrange2D.cpp provided by the instructor. Finally I used the ipython notebook plot\_Lagrange2D.ipynb provided by the instructor to graph the results outputted from test\_Lagrange2D.cpp.

The results are given in the graphs. Visually we cannot compare p10 to p20. If you look extremely closely you can see that p20’s error seems to be sharper than p10’s error. However in the last evaluation (numerically) we can clearly see that p10’s error is much larger than p20’s error.

As we interpolate a polynomial with more nodes, the error becomes less. This is natural because as the number of nodes increases, we should be able to create a more accurate model of the function. This is the reason why p10 has more error than p20.

Complexity of 2D lagrange:

I define n as the size of set x and define m as the size of set y. The Lagrange2D function has a nested loop, which loops over m, n times. within the nested loop, the lagrange basis is called twice and added to the function evaluation at that point (n, m). The two lagrange basis calsl loop over m and n respectively. This makes the algorithmic cost of the Lagrange2D function O(nm(nm)), which can be simplified to O(n^2 m^2).

Part 3

In part 3, I created a Runge\_uniform.cpp, which uses my Lagrange2D function from part 2 and the two-dimensional Runge function given in the instructions. In this, I interpolate evenly spaced nodes at different values and number of nodes. I also created a Runge\_Chebyshev.cpp function, which does the same as Runge\_uniform.cpp, but it interpolates the polynomial at the nodes given by the formula given in the instructions. I then created an ipython notebook named Runge2D.ipynb that loads Runge.txt, p6\_uni.txt, p24\_uni.txt, p6\_Cheb.txt and p24\_Cheb.txt using the numpy loadtxt command and then plots these into separate figure windows. It then computes the error and then graphs these errors.

The graph shows that as the number of interpolation nodes increased, the error decreased. It also shows that when we choose Chebyshev nodes instead of uniformly spaced nodes, the error also increased.

The best is increasing the number of interpolation nodes and using Chebyshev nodes. The second best is using Chebyshev nodes and not increasing the number of interpolation nodes. The second worst is using uniformly spaced nodes and increasing the number of interpolation nodes. The worst is using uniformly spaced nodes and not increasing the number of interpolation nodes. There is a point where increasing the amount of interpolating nodes does not decrease the error, and node selection becomes much more important. The 1D lagrange error is still meaningful because the 2D lagrange function is created by using the 1D lagrange. In the 2D lagrange, at every evaluation, the 1D lagrange of the x and the 1D lagrange of the y were multiplied together. so the error of the 2D lagrange is the error of the 1D lagrange of x multiplied by the error of the 1D lagrange of y. So f(x,y) –p(x,y) = E(x,y) = (f^(n+1)(c))/(n+1)! prod i=0 to n (x-xi) \* (f^(m+1)(d))/(m+1)! prod i=0 to m (y-yi). Where c lies in domain x and d lies in domain y.

Newton\_interpolant.cpp

#include <iostream>

#include "matrix.hpp"

Matrix Newton\_coefficients(Matrix& x, Matrix& y){ //returns vector of coefficients

if(x.Size()!= y.Size()){

std::cerr << "Newton error: the sizes of the two matrixes are not the same" << std::endl;

return Matrix(0,0);

}

else{

Matrix coef(x.Size());

for(int i = 0; i < coef.Size(); i++){

coef(i) = y(i);

}

for(int i = 1; i < coef.Size(); i++){

for(int a = coef.Size() - 1; a >= i; a--){

coef(a) = (coef(a) - coef(a-1))/(x(a) - x(a-i));

}

}

return coef;

}

}

double Newton\_evaluate(Matrix& x, Matrix& c, double z){ //returns the newton method evaluated using vector of coefficients and vector of x values

double temp = c(c.Size()-1);

for(int i = c.Size()-1; i >= 0; i--){

temp = temp \* (z-x(i)) + c(i);

}

return temp;

}

test\_Newton.cpp

//Van Tran

//11/9/15

//Hisci

// Inclusions

#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& y, double z);

// This routine tests the function Newton.cpp

int main(int argc, char\* argv[]) {

// simple Lambda function for f(x)

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); //CALCULATE THE COEFFICIENTS

// evaluate the Lagrange polynomial at the points z, storing in p

Matrix p(n);

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)));

}

} // end routine

test\_Newton.txt(output from test\_Newton.exe)

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 0

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 0

0.725 1.6053630078155 1.6053630078155 4.4e-16

0.775 1.8125438090848 1.8125438090848 0

0.825 2.0787033700036 2.0787033700036 4.4e-16

0.875 2.4201091598060 2.4201091598060 1.8e-15

0.925 2.8582584261957 2.8582584261957 1.2e-14

0.975 3.4218194055573 3.4218194055575 1.5e-13

Compare.cpp

//Van Tran

//11/9/15

//includes

#include <chrono>

#include <ctime>

#include <iostream>

#include <math.h>

#include "matrix.hpp"

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() {

std::chrono::time\_point<std::chrono::system\_clock> start, end;

Matrix xmatrix;

Matrix ymatrix;

Matrix zmatrix;

//lagrange

for(int n = 10; n <= 80; n=n\*2){ //repeat for n values

xmatrix = Linspace(-2.0, 2.0, n+1, 1); //Create a set of (n + 1) evenly-spaced nodes, x, over the interval [−2, 2].

ymatrix= Matrix(xmatrix.Size());

for(int i = 0; i < xmatrix.Size(); i++){ //Create the set of function data values where y(i) = cosh(x^2/3).

ymatrix(i) = cosh((xmatrix(i)\*xmatrix(i))/3);

}

for(int m = 100; m <= 100000; m=m\*10){ //repeat for m values

zmatrix = Linspace(-2.0, 2.0, m+1, 1); //Create a set of (m + 1) evenly-spaced evaluation points z over the interval [−2, 2].

start = std::chrono::system\_clock::now(); //Measure and print out the time it takes to evaluate {p(zi) from i=0 to m.

//Evaluate lagrange(x,y)

for(int i = 0; i < zmatrix.Size(); i++){

Lagrange(xmatrix, ymatrix, zmatrix(i));

}

end = std::chrono::system\_clock::now();

std::chrono::duration<double> elapsed\_seconds = end-start;

std::cout << "finished lagrange evaluation of " << m << " points using interpolating polynomial of " << n << " points in "<< elapsed\_seconds.count() << "s" << std::endl;

}

}

//newton

Matrix cmatrix;

for(int n = 10; n <= 80; n=n\*2){ //repeat for n values

xmatrix = Linspace(-2.0, 2.0, n+1, 1); //Create a set of (n + 1) evenly-spaced nodes, x, over the interval [−2, 2].

ymatrix = Matrix(xmatrix.Size());

for(int i = 0; i < xmatrix.Size(); i++){ //Create the set of function data values where y(i) = cosh(x^2/3).

ymatrix(i) = cosh(xmatrix(i)\*xmatrix(i)/3);

}

start = std::chrono::system\_clock::now();

cmatrix = Newton\_coefficients(xmatrix, ymatrix); //Create and time how long it takes to make coefficient matrix

end = std::chrono::system\_clock::now();

std::chrono::duration<double> elapsed\_seconds = end-start;

std::cout << "finished newton coefficient evaluation of " << n << " points in " << elapsed\_seconds.count() << "s" << std::endl;

for(int m = 100; m <= 100000; m=m\*10){ //repeat for m values

zmatrix = Linspace(-2.0, 2.0, m+1, 1); //Create a set of (m + 1) evenly-spaced evaluation points z over the interval [−2, 2].

start = std::chrono::system\_clock::now(); //Measure and print out the time it takes to evaluate {p(zi) from i=0 to m.

//Evaluate lagrange(x,y)

for(int i = 0; i < zmatrix.Size(); i++){

Newton\_evaluate(xmatrix, ymatrix, zmatrix(i));

}

end = std::chrono::system\_clock::now();

std::chrono::duration<double> elapsed\_seconds = end-start;

std::cout << "finished newton evaluation of " << m << " points using interpolating polynomial of " << n << " points in "<< elapsed\_seconds.count() << "s" << std::endl;

}

}

}

compare.txt(output for compare.exe)

finished lagrange evaluation of 100 points using interpolating polynomial of 10 points in 0.00215154s

finished lagrange evaluation of 1000 points using interpolating polynomial of 10 points in 0.02126s

finished lagrange evaluation of 10000 points using interpolating polynomial of 10 points in 0.214164s

finished lagrange evaluation of 100000 points using interpolating polynomial of 10 points in 1.47474s

finished lagrange evaluation of 100 points using interpolating polynomial of 20 points in 0.0025428s

finished lagrange evaluation of 1000 points using interpolating polynomial of 20 points in 0.0247934s

finished lagrange evaluation of 10000 points using interpolating polynomial of 20 points in 0.247851s

finished lagrange evaluation of 100000 points using interpolating polynomial of 20 points in 2.65719s

finished lagrange evaluation of 100 points using interpolating polynomial of 40 points in 0.03014s

finished lagrange evaluation of 1000 points using interpolating polynomial of 40 points in 0.299898s

finished lagrange evaluation of 10000 points using interpolating polynomial of 40 points in 1.23937s

finished lagrange evaluation of 100000 points using interpolating polynomial of 40 points in 9.43288s

finished lagrange evaluation of 100 points using interpolating polynomial of 80 points in 0.0372653s

finished lagrange evaluation of 1000 points using interpolating polynomial of 80 points in 0.368567s

finished lagrange evaluation of 10000 points using interpolating polynomial of 80 points in 3.69137s

finished lagrange evaluation of 100000 points using interpolating polynomial of 80 points in 36.9135s

finished newton coefficient evaluation of 10 points in 7.674e-06s

finished newton evaluation of 100 points using interpolating polynomial of 10 points in 4.5703e-05s

finished newton evaluation of 1000 points using interpolating polynomial of 10 points in 0.000461563s

finished newton evaluation of 10000 points using interpolating polynomial of 10 points in 0.00454996s

finished newton evaluation of 100000 points using interpolating polynomial of 10 points in 0.0457484s

finished newton coefficient evaluation of 20 points in 2.1882e-05s

finished newton evaluation of 100 points using interpolating polynomial of 20 points in 8.224e-05s

finished newton evaluation of 1000 points using interpolating polynomial of 20 points in 0.000815824s

finished newton evaluation of 10000 points using interpolating polynomial of 20 points in 0.0082393s

finished newton evaluation of 100000 points using interpolating polynomial of 20 points in 0.0818487s

finished newton coefficient evaluation of 40 points in 8.0324e-05s

finished newton evaluation of 100 points using interpolating polynomial of 40 points in 0.000155033s

finished newton evaluation of 1000 points using interpolating polynomial of 40 points in 0.00153996s

finished newton evaluation of 10000 points using interpolating polynomial of 40 points in 0.015522s

finished newton evaluation of 100000 points using interpolating polynomial of 40 points in 0.159127s

finished newton coefficient evaluation of 80 points in 0.000297698s

finished newton evaluation of 100 points using interpolating polynomial of 80 points in 0.000305311s

finished newton evaluation of 1000 points using interpolating polynomial of 80 points in 0.00302501s

finished newton evaluation of 10000 points using interpolating polynomial of 80 points in 0.0299103s

finished newton evaluation of 100000 points using interpolating polynomial of 80 points in 0.299735s

Lagrange2D.cpp

//Van Tran

//11/9/15

//includes

#include "matrix.hpp"

double Lagrange\_basis(Matrix& x, int i, double z);

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

double p = 0.0;

for(int i = 0; i<x.Size() ;i++){

for(int j = 0; j<y.Size() ;j++){

p += f(i, j) + Lagrange\_basis(x, i, a) \* Lagrange\_basis(y, j, b);

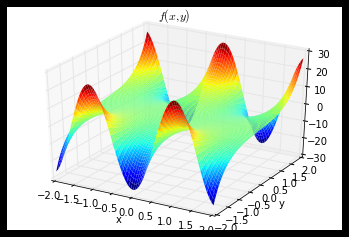
}

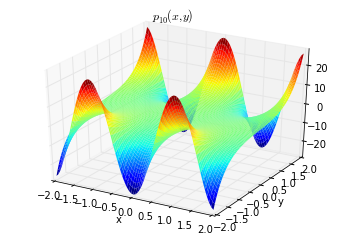
}

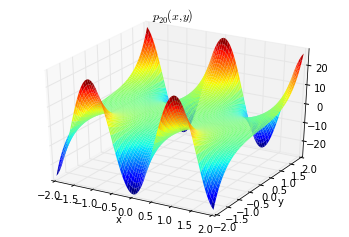
return p;

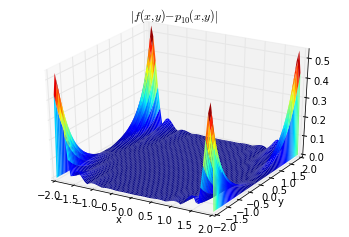
}

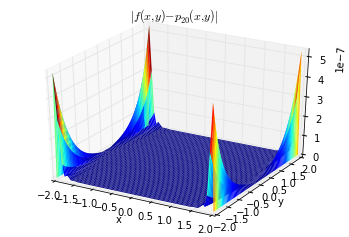
Graphs:











Runge\_uniform.cpp

//Van Tran

//11/9/15

//includes

#include "matrix.hpp"

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

int main(){

int n = 6;

int m = n;

Matrix x = Linspace(-4.0, 4.0, m+1, 1);

Matrix y = Linspace(-4.0, 4.0, n+1, 1);

Matrix f(x.Size(), y.Size());

for(int i = 0; i < x.Size(); i++){

for(int j = 0; j < y.Size(); j++){

f(i, j) = 1/(1+(x(i)\*x(i)) + (y(j)\*y(j)));

}

}

Matrix a = Linspace(-4.0, 4.0, 201, 1);

a.Write("avals.txt");

Matrix b = Linspace(-4.0, 4.0, 101, 1);

b.Write("bvals.txt");

Matrix p6(a.Size(), b.Size());

for(int i = 0; i < a.Size(); i++){

for(int j = 0; j < b.Size(); j++){

p6(i, j) = Lagrange2D(x, y, f, a(i), b(j));

}

}

p6.Write("p6\_uni.txt");

//do the same for n=24=m

n = 24;

m = n;

x = Linspace(-4.0, 4.0, m+1, 1);

y = Linspace(-4.0, 4.0, n+1, 1);

f = Matrix(x.Size(), y.Size());

for(int i = 0; i < x.Size(); i++){

for(int j = 0; j < y.Size(); j++){

f(i, j) = 1/(1+(x(i)\*x(i)) + (y(j)\*y(j)));

}

}

Matrix p24(a.Size(), b.Size());

for(int i = 0; i < a.Size(); i++){

for(int j = 0; j < b.Size(); j++){

p24(i, j) = Lagrange2D(x, y, f, a(i), b(j));

}

}

p24.Write("p24\_uni.txt");

Matrix runge(201, 101);

for(int i = 0; i < a.Size(); i++){

for(int j = 0; j < b.Size(); j++){

runge(i, j) = 1/(1+(a(i)\*a(i)) + (b(j)\*b(j)));

}

}

//create runge matrix

runge.Write("Runge.txt");

}

Runge\_Chebyshev.cpp

//Van Tran

//11/9/15

//includes

#include "matrix.hpp"

#include <cmath>

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

int main(){

const double pi = atan(1) \* 4;

int n = 6;

int m = n;

Matrix x(m +1);

Matrix y(n + 1);

for(int i = 0; i < x.Size(); i++){

x(i) = 4\*cos(((2\*i + 1)\*pi)/(2\*m + 2));

}

for(int i = 0; i < y.Size(); i++){

y(i) = 4\*cos(((2\*i + 1)\*pi)/(2\*m + 2));

}

Matrix f(x.Size(), y.Size());

for(int i = 0; i < x.Size(); i++){

for(int j = 0; j < y.Size(); j++){

f(i, j) = 1/(1+(x(i)\*x(i)) + (y(j)\*y(j)));

}

}

Matrix a = Linspace(-4.0, 4.0, 201, 1);

a.Write("avals.txt");

Matrix b = Linspace(-4.0, 4.0, 101, 1);

b.Write("bvals.txt");

Matrix p6(a.Size(), b.Size());

for(int i = 0; i < a.Size(); i++){

for(int j = 0; j < b.Size(); j++){

p6(i, j) = Lagrange2D(x, y, f, a(i), b(j));

}

}

p6.Write("p6\_Cheb.txt");

//do the same for n=24=m

n = 24;

m = n;

x = Linspace(-4.0, 4.0, m+1, 1);

y = Linspace(-4.0, 4.0, n+1, 1);

f = Matrix(x.Size(), y.Size());

for(int i = 0; i < x.Size(); i++){

for(int j = 0; j < y.Size(); j++){

f(i, j) = 1/(1+(x(i)\*x(i)) + (y(j)\*y(j)));

}

}

Matrix p24(a.Size(), b.Size());

for(int i = 0; i < a.Size(); i++){

for(int j = 0; j < b.Size(); j++){

p24(i, j) = Lagrange2D(x, y, f, a(i), b(j));

}

}

p24.Write("p24\_Cheb.txt");

}

Graphs:

