Project 4

Van Tran

Math 3316

12/1/15

Part 1

In part 1 I created a composite numerical integration function named composite_int using the 4 node Gaussian elimination nodes and weights given in the book. I mimicked the composite_Gauss2 function provided by the instructor and just changed the nodes and weights accordingly. These nodes and weights are calculated in the getconstants.cpp, which calculates the value of the nodes and weights with a double precision which I then plugged into my 4 node Gaussian elimination function. I did this to make my program more efficient. Although the big O does not change, it does remove a significant constant from the composite_int function. Although this may seem trivial, the functions created later call composite_int multiple times.

I chose to use the 4 Gaussian quadrature because the Gaussian Quadrature formulas have an exact approximation for polynomials of degree 2n + 1. the requirements of the rule is that the numerical integration is at least $O(h^8)$ accurate, and be from the book or an invention of my own. the 4 Gaussian quadrature formula is accurate for polynomials of degree 9 and is inside the book. Both requirements are met. The other option was to use the newton cotes quadrature formulas, however they are not as accurate nor converge as fast as their Gaussian quadrature counterparts.

I then created a main routine in the file test_int.cpp that emulates the routine test_Gauss2.cpp given by the instructor. I modified test_Gauss2.cpp by making it call my composite_int function I made instead of the composite_Gauss2, and messed around with the n values to better demonstrate my convergence rate.

I found that the function converges extremely fast and the convergence becomes smaller than floating point error when the number of subintervals gets to about 160. If we exceed this point the function becomes unpredictable, and the convergence often becomes negative or infinite.

Part 2

In part 2 I created an adaptive_int() function, which computes the integral of a function by calling my composite_int function until the result is within a given desired accuracy. Specifically, with a given atoll and rtol, I fulfilled the condition:

$|I(f) - R_n(f)| < \text{rtol}\ |I(f)| + \text{atol}$. where I(f) = the actual integral.

We are trying to approximate I(f), so obviously we do not have I(f), however, we do know that:

$|R_{n+k}(f) - R_n(f)| < |I(f) - R_n(f)|$ **where k > 0**

Because the difference between the two approximations is going to be less than the difference between the actual value and the approximation, as long as the approximations converge to I(f) as n increases (which is the case). We can merge these two equations to get

$|R_{n+k}(f) - R_n(f)| < |I(f) - R_n(f)| < \text{rtol}\ |I(f)| + \text{atol}$ **where k > 0**

and then take out the middle equation:

$|R_{n+k}(f) - R_n(f)| < \text{rtol}\,|I(f)| + \text{atol}$ **where k > 0**

In my adaptive_int function, I call the composite_int function with an initial n of 20 and set it to the "result" (n is the number of intervals). In a loop, I then multiply n by two and then call the composite_int function with the current n and set it to the "bounder". I then test whether the result is acceptable given the boundary and tolerances given by the user using the condition above. I then set the "bounder" to n+1 and called the composite_int function. then = n*2 and loop over until the condition given is true. However, I exit the program if n approaches 160 because the $I(f) - R_n(f)$ is smaller than the floating point error at that point., and will not converge anymore. During each step that I call the adaptive_int function I increment the counter Ntot (the total number of subintervals calculated) by the n used in the adaptive_int function call.

At first I decided to make the bounder of the previous loop the result of the next loop, in order to call composite_int less. However, this was a naïve idea, as the difference between the bounder and result would be too large since the differences in n were large between the bounder and result, and if I tried to make the n smaller between the bounder and the result, the amount of calls needed to converge to an accurate answer becomes extremely large. I then settled with the strategy described in the paragraph above.

I chose to make the n double each loop because of previous experience of a similar problem. In computer science, we often do not know how large a data structure may be (the number of elements is unknown). These data structures must "adapt" if their maximum capacity is reached. There are different techniques implemented in each data structure depending on how expensive it is to resize that data structure. A common practice many standard libraries use is to double the capacity of the data structure (stl vectors in c++ and java). I decided that the problem was similar enough for me to use this solution.

I then created a main routine in the file test_adapt.cpp that uses the adaptive_int() function that I created to integrate the function provided using the tolerances (rtol, atol) with $\text{atol}_i = \text{rtol}_i/1000$, and rtol = $\{10e{-}2 , 10e{-}4 , 10e{-}6 , 10e{-}8 , 10e{-}10 , 10e{-}12\}$. I then print out the values of $|I(f) - R(f)|$ and $\text{rtol}\,|I(f)| + \text{atol}$, as well as the n and Ntot.

The method achieves the desired relative errors and behaves as expected. The amount of subintervals calculated gets very large however. I find my adaptive_int function to be quite unproductive for the work that it does. However, it provides the information that the answer is within a certain tolerance and does not calculate the full number of intervals needed for answers that are only require a less accurate answer, but calculates more than it needs to for large cases of n.

Part 3:

in part 3 I created a file carbon.cpp that contains two functions, erf and carbon. The objective of these two functions is to calculate

$$C(x,t,T) = C_s - (C_s - C_0)\,\mathrm{erf}\left(\frac{x}{\sqrt{4t\,D(T)}}\right),$$

where

$$D(T) = 6.2 \times 10^{-7} \exp\left(-\frac{8 \times 10^4}{8.31T}\right),$$

and

$$\mathrm{erf}(y) = \frac{2}{\sqrt{\pi}} \int_0^y e^{-z^2}\,\mathrm{d}z.$$

The erf function evaluates the math erf function defined above and uses the adaptive_int function to evaluate the integral in the function. The only small difference from the form stated above is that I evaluate 2/sqrt(pi) in my getconstants.cpp and enter it as a constant in double precision.

The carbon function evaluates the carbon function given the three parameters in the function above. The function uses the erf function defined to evaluate the erf function in the written formula above. the only differences in the form stated above from my code is that I replace C_8 with 0.02, replace C_0 with 0.001 and pull out the 4 from the square root.

I then created a main routine in file test_carbon.cpp that creates an array of 400 evenly-spaced T values over the interval [800, 1200] K and outputs it to disk as the file Temp.txt. Then it Creates an array of 600 evenly-spaced t values from t = 1 second up to t = 48 hours and output this to disk as the file time.txt. Next it Creates a 400 × 600 array that contains C(0.002, t, T) and outputs this to disk as the file C2mm.txt. Then it Creates a 400 × 600 array that contains C(0.004, t, T) and output this to disk as the file C4mm.txt. Next it Creates an array of length 600 containing C(0.002, t, 800) and outputs this to disk as the file C2mm 800K.txt. Then the program repeats this to output the carbon concentrations for a 2 mm depth at 900K (C2mm 900K.txt), 1000K (C2mm 1000K.txt), 1100K (C2mm 24hour.txt) and 1200K (C2mm 1200K.txt). Next the program creates an array of length 600 containing C(0.004, t, 800) and outputs it to disk as the file C4mm 800K.txt. Finally it repeat this to output the carbon concentrations for a 4 mm depth at 900K (C4mm 900K.txt), 1000K (C4mm 1000K.txt), 1100K (C4mm 1100K.txt) and 1200K (C4mm 1200K.txt). All of these results are computed using the tolerances rtol = 10^{-11} and atoll = 10^{-15}. In each of these steps I used the instructor's matrix class

instead of the arrays, because the matrix class prints out the information in a format that allows the matplot library in python to graph the information.

I added a test.txt which is made up of many of C(0.003, 129600, T) for T = 0 to T = 1999. I did this in order to get a better idea of where my answer will lie for part 4.

My results were a little strange in the C4mm.txt and C2mm.txt. When both the time and Temperature are at its smallest values, I get more error than expected. As the time and temperature become larger, the values become more stable. This is because of the floating point error, and the differences between the smaller values are less than the precision given in a double. It isn't noticeable in the contour plots, but is important to keep in mind.

Next I created ipython notebooks to plot the text files I just outputted. I created a filled contour plot of C(0.002, t, T) and C(0.004, t, T). I created a third figure with the curves for the carbon concentrations at a 2 mm depth, for the temperatures 800, 900, 1000, 1100 and 1200 Kelvin as a function of time overlaid on one another (5 line plots overlaid in different colors, containing my data from the files C2mm 800K.txt, etc.). I created a fourth figure with the curves for the carbon concentrations at a 4 mm depth, for the temperatures 800, 900, 1000, 1100 and 1200 Kelvin as a function of time overlaid on one another (5 line plots overlaid in different colors, containing my data from the files C4mm 800K.txt, etc.). I failed to do the tick marks as described in the instructions. The function the instructor provided did not work in my ipython notebook probably because my libraries are automatically updated by canopy and the documentation changed. The graphs still turned out and represented the data correctly.

My graphs turned out well, and look feasible for the most part. We can see that both the contour plots and line graphs for C2mm and C4mm look similar. In both cases it seems as if the C2mm graphs are shifted a little up compared to the C4mm graphs. The C4mm graphs also look like they have less concavity then the C2mm graphs.

Part 4:

In part 4 I created a main routine that uses my carbon function and the bisection function I defined in project 2. The bisection function is a root finding function that returns the root given a function, the interval which the root lies, and the tolerance. The main routine's purpose is to solve the following problem:

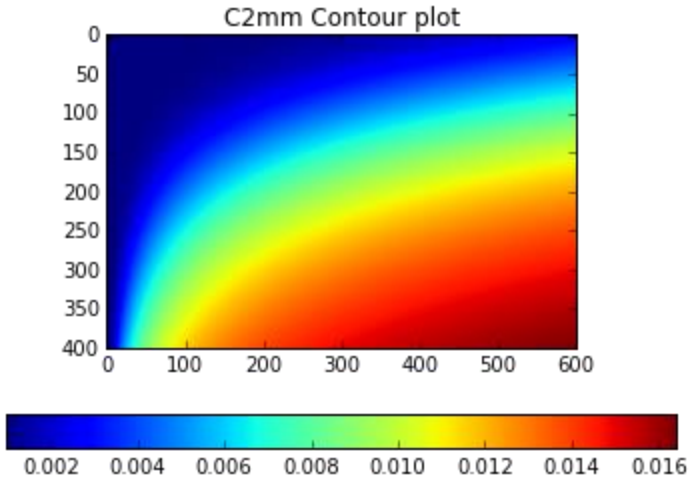
> *Determine the temperature, accurate to 0.0001 K, at which a depth of 3.0 mm in the steel will reach a carbon concentration of 0.6% under carburization at* $t = 36$ *hours.*

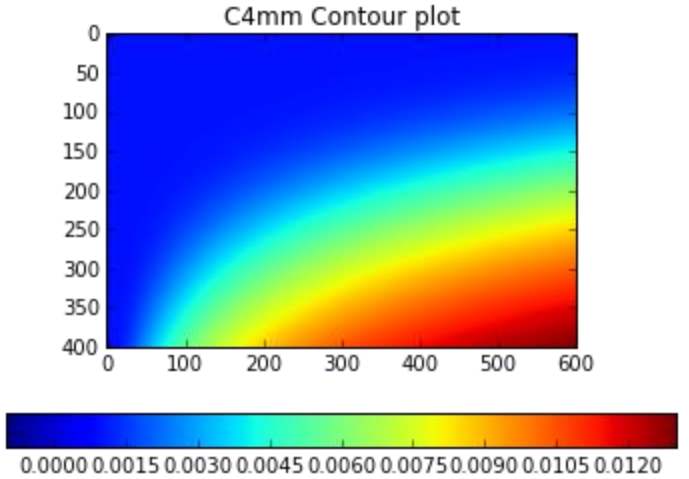
The problem asks you to solve for T in the equation C(.003, 129600, T) = 0.006 (the numbers are different because I converted the units in the problem to the units that the carbon function is specified to be). To be used in the bisection solver, I create a function where

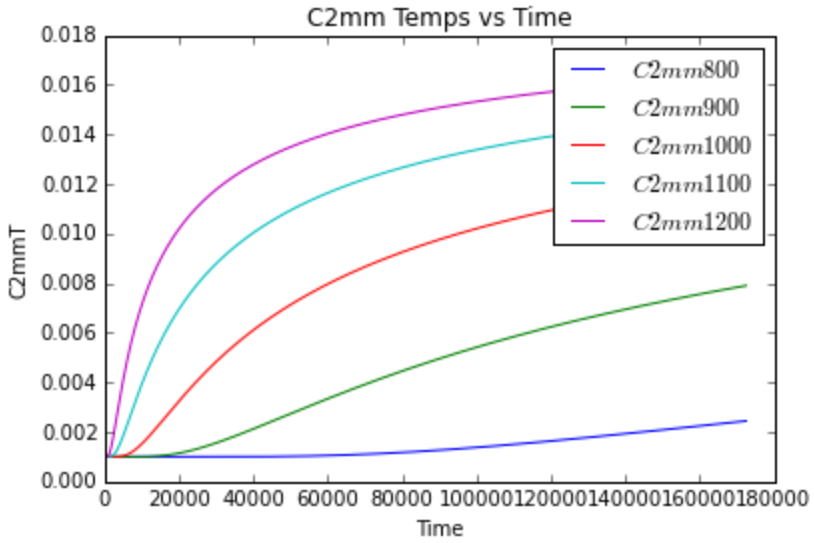
C (.003, 129600,T) – 0.006 and the bisection method solves for the root (zero) of the function.

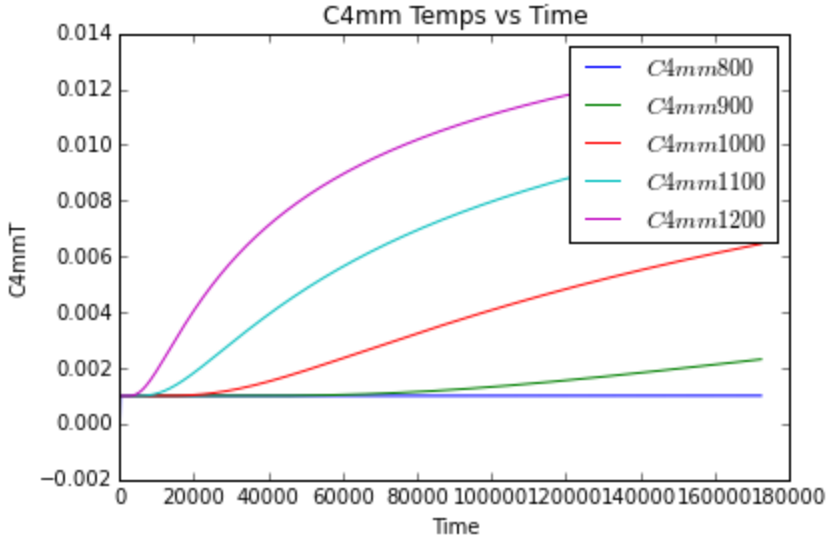
The answer I got is 961.26409681901213844 Kelvin. I plugged it back into the function just to make sure, but the answer is 0.0096579950252694714, which is not 0, but it is extremely close to it.

This answer makes sense because 0.003 is right between the two depths shown. This can be seen in the contour plots provided and are even clearer in the line graphs. I can tell from eyeballing the graph that the Temperature is going to be between 700 and 1100, however, I made the bisection go from 671 to 2000 just to be safe.


C2mm Contour plot
0
50
100
150
200
250
300
350
400
0
100
200
300
400
500
600
0.002
0.004
0.006
0.008
0.010
0.012
0.014
0.016


C4mm Contour plot
0
50
100
150
200
250
300
350
400
0
100
200
300
400
500
600
0.0000
0.0015
0.0030
0.0045
0.0060
0.0075
0.0090
0.0105
0.0120


C2mm Temps vs Time
C2mmT
Time
$C2mm800$
$C2mm900$
$C2mm1000$
$C2mm1100$
$C2mm1200$


C4mm Temps vs Time
C4mmT
Time
$C4mm800$
$C4mm900$
$C4mm1000$
$C4mm1100$
$C4mm1200$

getconstants.cpp

```
/*Author: Van Tran

getconstants.cpp

11/25/15*/

#include <iostream>

#include <math.h>

//includes


using namespace std;

int main(){//gives me various constants in floating point precision so I can
plug into programs for later

	cout.precision(17);

	cout << fixed << 128.0/225.0 << endl;

	cout << fixed <<(1.0/3.0) * sqrt(5.0 - 2.0*sqrt(10.0/7.0)) << endl;

	cout << fixed <<(1.0/3.0) * sqrt(5.0 + 2.0*sqrt(10.0/7.0)) << endl;

	cout << fixed <<(322.0 + 13.0*(sqrt(70.0)))/900.0 << endl;

	cout << fixed <<(322.0 - 13.0*(sqrt(70.0)))/900.0 << endl;

	cout << fixed << (2.0/sqrt(3.141592653589793238463)) << endl;

	return 0;

}
```

composite_int.cpp

```
*Author: Van Tran

composite_int.cpp

11/25/15*/

#include <iostream>

#include <math.h>

#include "fcn.hpp"

//includes

using namespace std;


//gaussian quadrature rule 4 for function f

double composite_int(Fcn& f, const double a, const double b, const int n){

  // check input arguments

  if (b < a) {

    cerr << "error: illegal interval, b < a\n";

    return 0.0;

  }

  if (n < 1) {

    cerr << "error: illegal number of subintervals, n < 1\n";

    return 0.0;

  }

   // set subinterval width

  double h = (b-a)/n;


  // set nodes/weights defining the quadrature method (within each
subinterval)

  //gotten from getroots.cpp

  double x1 =0.0;

  double x2 = -0.53846931010568300;

  double x3 = 0.53846931010568300;

  double x4 = -0.90617984593866396;
```

```
  double x5 = 0.90617984593866396;

  double w1 =0.56888888888888889;
  double w2 =0.47862867049936647;
  double w3 =0.47862867049936647;
  double w4 =0.23692688505618908;
  double w5 =0.23692688505618908;

  // initialize result
  double F = 0.0;

  // loop over subintervals, accumulating result
  double xmid, node1, node2, node3, node4, node5;
  for (int i=0; i<n; i++) {

    // determine evaluation points within subinterval (basically scale so the
nodes in gaussian rule are nodes in subinterval)
    xmid  = a + (i+0.5)*h;
    node1 = xmid + .5*h*x1;
    node2 = xmid + .5*h*x2;
    node3 = xmid + .5*h*x3;
    node4 = xmid + .5*h*x4;
    node5 = xmid + .5*h*x5;

    // add Gauss4 approximation on this subinterval to result
    F += w1*f(node1) + w2*f(node2) + w3*f(node3) + w4*f(node4) + w5*f(node5);

  } // end loop

  // return final result
  return (0.5*h*F);
```

```
}
```

test_int.cpp

```
/*Author: Van Tran
test_int.cpp
11/25/15*/
// Inclusions
#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include <vector>
#include <math.h>
#include "fcn.hpp"

using namespace std;

// function prototypes
double composite_int(Fcn& f, const double a, const double b, 13toll13int n);

// Integrand
class fcn : public Fcn {
public:
  double c, d;
  double operator()(double x) {   // function evaluation
    return (exp(c*x) + sin(d*x));
  }
  double antiderivative(double x) { // function evaluation
    return (exp(c*x)/c - cos(d*x)/d);
  }
};
```

```
// This routine tests the Gauss-8 method on a simple integral
int main(int argc, char* argv[]) {

  // limits of integration
  double a = -3.0;
  double b = 5.0;

  // integrand
  fcn f;
  f.c = 0.5;
  f.d = 25.0;

  // true integral value
  double Itrue = f.antiderivative(b) - f.antiderivative(a);
  printf("\n True Integral = %22.16e\n", Itrue);



  // test the Gauss-4 rule
  cout << "\n Gauss-5 approximation:\n";
  cout << "     n             R(f)            relerr    conv rate\n";
  cout << "  ------------------------------------------------\n";
  vector<int> n = {20, 40, 60, 80, 100, 120, 140, 160, 161, 162, 163, 164,
165, 166, 167, 168, 169, 170, 175, 176, 177, 178, 179};
  vector<double> errors(n.size());
  vector<double> hvals(n.size());

  // iterate over n values, computing approximations, error, convergence rate
  double Iapprox;
  for (int i=0; i<n.size(); i++) {

    printf("   %6i", n[i]);
```

```
    Iapprox = composite_int(f, a, b, n[i]);
    errors[i] = fabs(Itrue-Iapprox)/fabs(Itrue);
    hvals[i] = (b-a)/n[i];
    if (I == 0)
      printf("  %22.16e  %7.1e     ----\n", Iapprox, errors[i]);
    else
      printf("  %22.16e  %7.1e   %f\n", Iapprox, errors[i],
           (log(errors[i-1]) - log(errors[i]))/(log(hvals[i-1]) -
log(hvals[i])));

  }
  cout << "  -----------------------------------------------------\n";

}
```

adaptive_int.cpp

```
/*Author: Van Tran

adaptive_int.cpp

11/25/15*/

#include <iostream>

#include <math.h>

#include "fcn.hpp"

//includes

double composite_int(Fcn& f, const double a, const double b, 16toll16int n);

//f is function, a is lower bound of integral, b is 16toll bound of integral,
rtol is, 16toll is, R is the double that the result is stored in, n is the
int that shows the number of subintervals, Ntot is the total number of
intervals used along the way

int adaptive_int(Fcn& f, const double a, const double b, const double rtol,
const double 16toll, double& R, int& n, int& Ntot){

      double bounder;

      int I = 20; //current number of intervals

      int z = 0; //total number of intervals

      double result = composite_int(f, a, b, i);

      z = z+I;

      while(I <= 80){ //never let loop call composite_int with I > 160

            I = i*2;

            result = composite_int(f,a,b, i);

            z = z+I;

            bounder = composite_int(f, a, b, i+1);

            z =z+i+1;

            if(fabs(result - bounder) < rtol*fabs(bounder) + 16toll){

                  n = I;

                  Ntot = z;

                  R = result;

                  return 1; //success!

            }
```

```
	}

	n = I;

	Ntot = z;

	R = result;

	return 0; //failure because R(f) does not satisfy its boundary
conditions

}
```

test_adapt.cpp

```
/*Author: Van Tran

test_adapt.cpp

11/25/15*/

#include "fcn.hpp"

#include <math.h>

#include <iostream>

#include <stdio.h>

#include <stdlib.h>

//includes


using namespace std;


class fcn : public Fcn {

public:

  double c, d;

  double operator()(double x) {   // function evaluation

    return (exp(c*x) + sin(d*x));

  }

  double antiderivative(double x) { // function evaluation

    return (exp(c*x)/c - cos(d*x)/d);

  }

};


int adaptive_int(Fcn& f, const double a, const double b, const double
rtol,const double 18toll, double& R, int& n, int& Ntot);


int main(){//tests the adaptive_int function defined in adaptive_int.cpp

      double rtol[] = {10e-2, 10e-4, 10e-6, 10e-8, 10e-10, 10e-12};

      double 18toll[] = {rtol[0]/1000.0, rtol[1]/1000.0, rtol[2]/1000.0,
rtol[3]/1000.0 , rtol[4]/1000.0, rtol[5]/1000.0};
```

```
  // limits of integration
      double a = -3.0;
      double b = 5.0;

      // integrand
      fcn f;
      f.c = 0.5;
      f.d = 25.0;
      double R;
      int n;
      int Ntot;
      double Itrue = f.antiderivative(b) – f.antiderivative(a);
      printf("\n True Integral = %22.16e\n", Itrue);
      int success;
      cout.precision(17); //so prints without roundoff
      for(int I = 0; I < 6; i++){
          success = adaptive_int(f, a, b, rtol[i], 19toll[i], R, n, Ntot);
          if(success){
            cout << "SUCCESS" << endl;
          }
          cout << "Result: " << fixed << R << endl;
          cout << scientific << "final number of subintervals: " << n << endl
<< "total number of subintervals calculated: " << Ntot << endl << "|I(f) –
R(f)|: " << fabs(Itrue – R) << endl << "rtol|I(f)| + 19toll: " << rtol[i] *
fabs(Itrue) + 19toll[i] << endl << endl;
      }
}
```

carbon.cpp

```
/*Author: Van Tran

Carbon.cpp

11/25/15*/

#include <iostream>

#include <math.h>

#include "fcn.hpp"

//includes

int adaptive_int(Fcn& f, const double a, const double b, const double rtol,
const double atol, double& R, int& n, int& Ntot); //function declaration from
adaptive_int.cpp


class fcn : public Fcn {

public:

  double operator()(double x) {   //integral function

    return pow(2.71828182845904523536, (-1 * pow(x, 2)));

  }

};


fcn err;


double erfs(const double y, const double rtol, const double atol){

      double R = 0;

      int Ntot = 0, n = 0;

      return (1.12837916709551256 * R); //2/sqrt(pi) * integral

}


double carbon(const double x, const double t, const double T, const double
rtol, const double atol){

      double Dt = 6.2e-7 * exp(-8e4/(8.31*T));

      double errorf = erfs(x/(2*sqrt(t*Dt)), rtol, atol);

      return (0.02 - (0.019 * errorf));
```

```
}
```

test_carbon.cpp

```
/*Author: Van Tran

test_carbon.cpp

11/25/15*/

#include <fstream>

#include <iostream>

#include <iomanip> //for setprecision

#include "matrix.hpp"

//includes

double carbon(const double x, const double t, const double T, const double
rtol, const double atol); //function declaration from carbon.cpp

using namespace std;

int main(){

      Matrix T(400);

      for(int i = 0; i < 400; i++){ //Temptxt

            T(i) = i + 800;

      }

      T.Write("Temp.txt");

      Matrix t(600);

      double h = 172799.0/600.0;

      for(int i = 0; i < 600; i++){//timetxt

            t(i) = 1+ i * h;

      }

      t.Write("time.txt");

      Matrix C2mm(400, 600);

      Matrix C4mm(400, 600);

      for(int i = 0; i < 400; i++){ //C2mmtxt C4mmtxt

            for(int j = 0; j < 600; j++){

                  C2mm(i, j) = carbon(0.002, t(j), T(i), 10e-11, 10e-15);
```

```
            C4mm(i, j) = carbon(0.004, t(j), T(i), 10e-11, 10e-15);
      }
}
C2mm.Write("C2mm.txt");
C4mm.Write("C4mm.txt");
Matrix test(2000);
for(int i = 0; i < 2000; i++){
      test(i) = carbon(0.003, 129600, i, 10e-11, 10e-15);
}
test.Write("test.txt");

Matrix C2800;
Matrix C2900;
Matrix C21000;
Matrix C21100;
Matrix C21200;

Matrix C4800;
Matrix C4900;
Matrix C41000;
Matrix C41100;
Matrix C41200;

for(int i = 0; i < 600; i++){ //everything else
      C2800(i) = carbon(0.002, t(i), 800, 10e-11, 10e-15);
      C2900(i) = carbon(0.002, t(i), 900, 10e-11, 10e-15);
      C21000(i) = carbon(0.002, t(i), 1000, 10e-11, 10e-15);
      C21100(i) = carbon(0.002, t(i), 1100, 10e-11, 10e-15);
```

```
        C21200(i) = carbon(0.002, t(i), 1200, 10e-11, 10e-15);


        C4800(i) = carbon(0.004, t(i), 800, 10e-11, 10e-15);
        C4900(i) = carbon(0.004, t(i), 900, 10e-11, 10e-15);
        C41000(i) = carbon(0.004, t(i), 1000, 10e-11, 10e-15);
        C41100(i) = carbon(0.004, t(i), 1100, 10e-11, 10e-15);
        C41200(i) = carbon(0.004, t(i), 1200, 10e-11, 10e-15);
    }
    C2800.Write("C2mm_800K.txt");
    C2900.Write("C2mm_900K.txt");
    C21000.Write("C2mm_1000.txt");
    C21100.Write("C2mm_24hour.txt");
    C21200.Write("C2mm_1200.txt");


    C4800.Write("C4mm_800K.txt");
    C4900.Write("C4mm_900K.txt");
    C41000.Write("C4mm_1000.txt");
    C41100.Write("C4mm_1100.txt");
    C41200.Write("C4mm_1200.txt");

}
```

carbon.ipynnb(just the code)

Van Tran

Math 3316

11/26/15

Load all data

```
Temp = loadtxt('temp.txt')

Time = loadtxt('time.txt')

C2mm = loadtxt('C2mm.txt')

C4mm = loadtxt('C4mm.txt')

C2mm800 = loadtxt('C2mm_800K.txt')

C2mm900 = loadtxt('C2mm_900K.txt')

C2mm1000 = loadtxt('C2mm_1000.txt')

C2mm1100 = loadtxt('C2mm_24hour.txt')

C2mm1200 = loadtxt('C2mm_1200.txt')

C4mm800 = loadtxt('C4mm_800K.txt')

C4mm900 = loadtxt('C4mm_900K.txt')

C4mm1000 = loadtxt('C4mm_1000.txt')

C4mm1100 = loadtxt('C4mm_1100.txt')

C4mm1200 = loadtxt('C4mm_1200.txt')

figure()

imshow(C2mm)

colorbar(orientation='horizontal')

title('C2mm Contour plot')

figure()

imshow(C4mm)

colorbar(orientation='horizontal')

title('C4mm Contour plot')
```

```
plot(Time, C2mm800, Time, C2mm900, Time, C2mm1000, Time, C2mm1100, Time,
C2mm1200)

title('C2mm Temps vs Time')

xlabel('Time')

ylabel('C2mmT')

legend(('$C2mm800$', '$C2mm900$', '$C2mm1000$', '$C2mm1100$', '$C2mm1200$'))

plot(Time, C4mm800, Time, C4mm900, Time, C4mm1000, Time, C4mm1100, Time,
C4mm1200)

title('C4mm Temps vs Time')

xlabel('Time')

ylabel('C4mmT')

legend(('$C4mm800$', '$C4mm900$', '$C4mm1000$', '$C4mm1100$', '$C4mm1200$'))
```

application.cpp

```
/*Author: Van Tran

application.cpp

11/25/15*/

#include "fcn.hpp"

#include <iostream>

//includes

double bisection(Fcn& f, double a, double b, int maxit, double tol,bool
show_iterates); //used to solve root

double carbon(const double x, const double t, const double T, const double
rtol, const double atol); //used in function declaration

//function declarations

using namespace std;


int main(){

      class fcn : public Fcn { //define abstract class

      public:

            double operator()(double x) {   // function evaluation

            return (carbon(0.003, 129600, x, 10e-14, 10e-15) - 0.006);

            }

      };

      fcn f;

      cout.precision(17);

      cout << fixed << "The Temperature is: " << bisection(f, 671.0, 2000,
2000, 10e-14, 1) << "K" <<endl; //solve and print value solved

      cout << "start" << carbon(0.002, 129600, 961.26409681901213844, 10e-11,
10e-15) <<endl; //test whether value is correct

}
```

bisection.cpp

```
/* Daniel R. Reynolds
   SMU Mathematics
   Math 3316
   16 September 2015 */

// Inclusions
#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include <math.h>
#include "fcn.hpp"

using namespace std;

// This routine uses the bisection method to approximate a root of
// the scalar-valued nonlinear equation f(x)=0 to a solution
// tolerance of tol.
//
// Usage: c = bisection(f, a, b, maxit, tol, show_iterates);
//
// inputs:   f              user-supplied Fcn object
//           a              lower-bound on solution interval [double]
//           b              upper-bound on solution interval [double]
//           maxit          maximum allowed iterations [int]
//           tol            solution tolerance [double]
//           show_iterates  display/hide iteration information [bool]
// outputs:  c              approximate solution [double]
//
double bisection(Fcn& f, double a, double b, int maxit,
```

```
            double tol, bool show_iterates) {

  // initialize approximate solution
  double c = 0.5*(a+b);

  // check input arguments
  if (maxit < 1) {
    cerr << "warning: maxit = " << maxit << " < 1. Resetting to 100\n";
    maxit = 100;
  }
  if (a >= b) {
    cerr << "error: illegal interval [" << a << ", " << b << "]!\n";
    return 0.0;
  }
  if (tol < 1.e-15) {
    cerr << "warning: tol is too small, resetting to 1e-15\n";
    tol = 1.e-15;
  }

  // get initial function values, check whether root exists in interval
  double fa = f(a);
  double fb = f(b);
  if (fa*fb > 0.0) {
    cerr << "error: illegal interval, f(a)=" << fa << ", f(b)=" << fb <<
endl;
    return 0.0;
  }

  // begin iteration
  double fc, err=0.5*(b-a);
  cout << endl << " Bisection Method: initial |(b-a)/2| = " << err << endl;
```

```
  for (int i=0; i<maxit; i++) {

    // evaluate function at c
    fc = f(c);

    // update interval
    if (fa*fc < 0.0) {
      b = c;
      fb = fc;
    } else {
      a = c;
      fa = fc;
    }

    // compute updated guess, function value
    c = 0.5*(a+b);

    // check for convergence and output diagnostics
    err = 0.5*(b-a);
    if (show_iterates)
      cout << "   iter " << i << ", [" << a << "," << b
         << "], |(b-a)/2| = " << err << endl;
    if (err < tol)  break;

  } // end loop

  // return final result
  return c;

} // end of function
```

```
/* Daniel R. Reynolds

   SMU Mathematics

   16 September 2015


   This file defines a "fcn" class, that essentially creates a small object
to

   evaluate a user-defined function that carries along its own data
parameters. */


#ifndef FCN_DEFINED__

#define FCN_DEFINED__




// Fcn class

class Fcn {


public:


  // pure virtual function makes this an 'abstract' base class

  virtual double operator()(double x) = 0;


};


#endif  // FCN_DEFINED__
```

Makefile

```
#######################################################################

#  Makefile for project 4

#

#  Daniel R. Reynolds

#  SMU Mathematics

#  Math 3316

#  31 October 2015

#######################################################################


# compiler & flags

CXX = g++

CXXFLAGS = -O2 -std=c++11


# makefile targets

all : test_Gauss2.exe test_int.exe test_adapt.exe getconstants.exe
test_carbon.exe application.exe


test_Gauss2.exe : test_Gauss2.cpp composite_Gauss2.cpp

                                                                    $(CXX)
$(CXXFLAGS) $^ -o $@


test_int.exe : test_int.cpp composite_int.cpp

                                                                    $(CXX)
$(CXXFLAGS) $^ -o $@


test_adapt.exe : test_adapt.cpp adaptive_int.cpp composite_int.cpp

                                                                    $(CXX)
$(CXXFLAGS) $^ -o $@
```

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

application.exe : application.cpp carbon.cpp adaptive_int.cpp
composite_int.cpp bisection.cpp fd_newton.cpp
	$(CXX) $(CXXFLAGS) $^ -o $@

test_carbon.exe : test_carbon.cpp carbon.cpp adaptive_int.cpp
composite_int.cpp matrix.cpp
	$(CXX) $(CXXFLAGS) $^ -o $@

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

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


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