Project 4: Numerical Integration

Erik Gabrielsen

Math 3316

December 2, 2015

Overview:

In this Project, I designed a function that would approximate an integral using a method of my choosing as well as a test function to ensure that it works correctly and efficiently. I will then incorporate this function into an adaptive Numerical Integration that requests to find a result of a desired accuracy using the least possible computational effort. I will then apply these two methods to model the concentrations of carbon relative to iron in an alloy during the process of carburizing. Finally, I use a root finding function to find an approximation of the temperature of carbon need to reach a certain carbon concentration at a certain time.

Part I – High-Order Numerical Integration

In Part I, I was provided a composite Gaussian numerical integration formula with 2 nodes that approximates a function over an integral [a, b] with a convergence of $O(h^4)$. My task is to create a composite numerical integration routine that approximates the integral with a convergence of $O(h^8)$ in a function with the following signature:

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

My first thought was to add one additional node to the original 2 node Gaussian method. This increased the convergence but only to a convergence of $O(h^6)$. Therefore, I used 4 nodes in my version of the Gaussian numerical integration. To test the convergence of this routine I created a file (test_int.cpp) which would run the function several times with different values of n nodes to ensure that it converges correctly. Using 10 different n values in the range [20, 200] I determined that the method converges at a rate of $O(h^8)$ as desired. Here is the output from the test routine:

```
┌─[erikgabrielsen][Eriks-MacBook-Air][±][Project4 ?:7 ✗][2.0.0][~/D
└─▪ ./test_int.exe

True Integral = 2.3924089071413306e+01

Comp Int approximation:
    n            R(f)            relerr    conv rate
 ------------------------------------------------
     20  2.3924921763521915e+01  3.5e-05     ----
     40  2.3924084859071119e+01  1.8e-07   7.627017
     60  2.3924088999577471e+01  3.0e-09   10.041285
     80  2.3924089065582557e+01  2.4e-10   8.729213
    100  2.3924089070518338e+01  3.7e-11   8.398689
    120  2.3924089071214610e+01  8.3e-12   8.254705
    140  2.3924089071356974e+01  2.4e-12   8.177176
    160  2.3924089071394299e+01  7.9e-13   8.136300
    180  2.3924089071405966e+01  3.1e-13   8.078262
    200  2.3924089071410169e+01  1.3e-13   8.067960
 ------------------------------------------------
```

As the number of nodes increased, the approximation increased in accuracy, as expected as you are reducing the amount of error during each subinterval. I chose to solve solve for the eight optimal evaluation points myself and incorporate the values as constants in order to save computation time in calculating the approximation of the integral, rather than reevaluating these constants every time the function is called.

Code: *composite_int.cpp*
test_int.cpp

Part II – Adaptive Numerical Integration

In Part II, I was to design an adaptive numerical integration function called *adaptive_int.cpp* that will compute the integrand of a function along the interval [a, b] by calling my *composite_int()* function in an adaptive manner. The goal is to create such a routine so that the result uses the least possible computational effort. I will achieve this by minimizing n, but not working overtime to do so. I then created a test function called *test_adapt.cpp* that uses my *adaptive_int()* function to integrate the same problem as in part I but this time stopping when the increase in accuracy is less than a combination of given relative and absolute tolerances. By doing this, the routine will refrain from under-guessing or over-guessing the correct value of n. I now used the tolerance with $atol_i = rtol_i/1000$, and rtol = $\{10^{-2}, 10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}\}$. My method does achieve the desired relative error at each tolerance level, however when the relative tolerance reaches 10^{-10} the approximation becomes too accurate as the value of n doubles and over-guesses the correct value of n. Here are the results of the *test_adapt.cpp* routine:

```
┌─[erikgabrielsen][Eriks-MacBook-Air][±][Project4 U:1 ?:7 ✗][2.0.0][~/Desktop/MATH3316/Proje
└─▪ ./test_adapt.exe
True integral value = 6.5033551341673331e+01

    rtol     atol      n    nTot        R         |I(f)-R(f)|   rtol|I(f)|+atol     Passed
  1.0e-02 | 1.0e-05 |  20 |    35 | 6.503599e+01 |  2.4e-03  |   6.5e-01      |      yes
  1.0e-04 | 1.0e-07 |  40 |    95 | 6.503354e+01 |  1.2e-05  |   6.5e-03      |      yes
  1.0e-06 | 1.0e-09 |  80 |   215 | 6.503355e+01 |  1.7e-08  |   6.5e-05      |      yes
  1.0e-08 | 1.0e-11 | 160 |   455 | 6.503355e+01 |  5.6e-11  |   6.5e-07      |      yes
  1.0e-10 | 1.0e-13 | 320 |   935 | 6.503355e+01 |  2.0e-13  |   6.5e-09      |      yes
  1.0e-12 | 1.0e-15 | 320 |   935 | 6.503355e+01 |  2.0e-13  |   6.5e-11      |      yes
```

For this adaptive method I chose to update the value of n by a value k which would be a proportion of the current n value and itself. This way, n never becomes too big to where the value becomes over-guessed or under-guessed. For the above version I began with an initial k=5. To prevent R from over working I reduced the initial value of k to 4. This created the following output, which does not overwork for finding R:

```
┌─[erikgabrielsen][Eriks-MacBook-Air][±][Project4 U:1 ?:5 ✗][2.0.0][~/Desktop/MATH3316/Proje
└─▪ ./test_adapt.exe
True integral value = 6.5033551341673331e+01

    rtol      atol      n     nTot        R          |I(f)-R(f)|   rtol|I(f)|+atol      Passed
  1.0e-02 | 1.0e-05 |  19 |    34 | 6.503761e+01 |   4.1e-03   |    6.5e-01       |      yes
  1.0e-04 | 1.0e-07 |  38 |    91 | 6.503353e+01 |   2.4e-05   |    6.5e-03       |      yes
  1.0e-06 | 1.0e-09 |  76 |   205 | 6.503355e+01 |   2.6e-08   |    6.5e-05       |      yes
  1.0e-08 | 1.0e-11 | 152 |   433 | 6.503355e+01 |   8.5e-11   |    6.5e-07       |      yes
  1.0e-10 | 1.0e-13 | 304 |   889 | 6.503355e+01 |   3.3e-13   |    6.5e-09       |      yes
  1.0e-12 | 1.0e-15 | 608 |  1801 | 6.503355e+01 |   2.8e-14   |    6.5e-11       |      yes
```

As predicted, the number of nTot and n is reduced for each iteration. However, I did notice that if I were to include more tolerance levels that this would cause my routine to over guess in some instances and under guess in others. Therefore choosing the right value of k to increase n by each time is almost impossible to distinguish for all cases and has to be chosen based on the tolerance levels that one is testing. Maybe there is one value of k that can be used for all tolerance levels that I just don't know about but in my trials I found no such number.

Code: *adaptive_int.cpp*
test_adapt.cpp

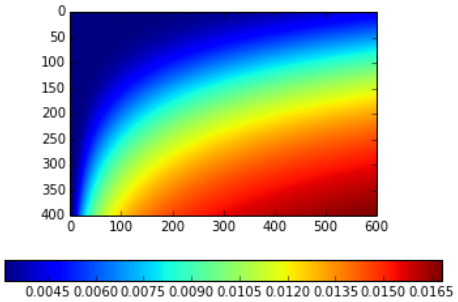
Part III – Application

In Part III, I will use parts I and II in order to evaluate an equation used in carburizing, the hardening of steel by increasing the concentration of carbon in relative to the iron in the alloy. To do this I created a file *carbon.cpp* that includes the two functions:

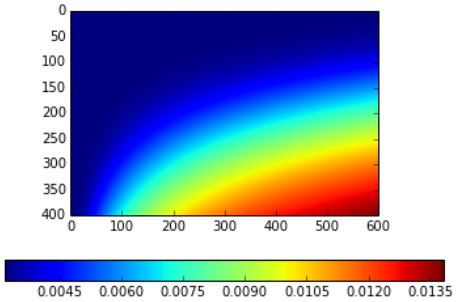
```
double erf(const double y, const double rtol, const double
          atol);
double carbon(const double x, const double t, const double T,
          const double rtol, const double atol);
```

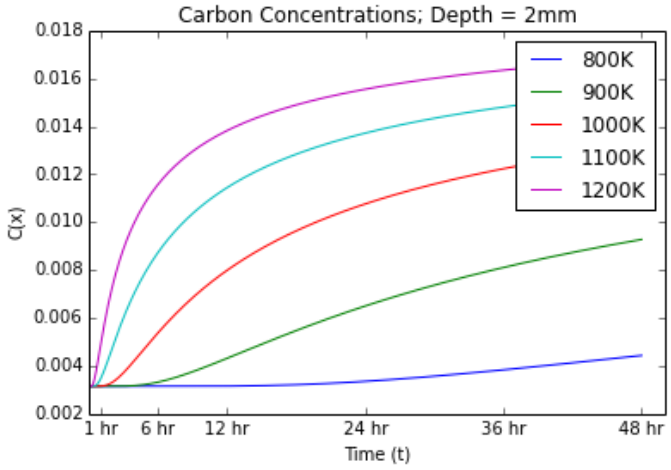
The first function *erf()* is the error function and uses my *adaptive_int()* function to determine the concentration of carbon *C(x, t, T)* which is represented by the second function *carbon()*. I also added a function *D(const double T)* which is used to find the temperature-dependent diffusion coefficient of the steel that is used in *carbon()*. I then created a test file *test_carbon.cpp* that creates two 400 x 600 arrays representing *C()* at a depth of 2mm and at 4mm. I then created ten arrays of length 600 to represent the concentration of carbon at a depth of 2 and 4mm at specified temperatures. Those temperatures were K = {800, 900, 1000, 1100, 1200} Kelvin. All of these results were computed using tolerances of rtol = 10^{-11} and atol = 10^{-15}. I write all of the arrays to disk and then load them in a python script to give a graphical representation of the data I collected. Here are the graphs I produced by my python book:

Carbon with a depth of 2mm



Carbon with a depth of 4mm





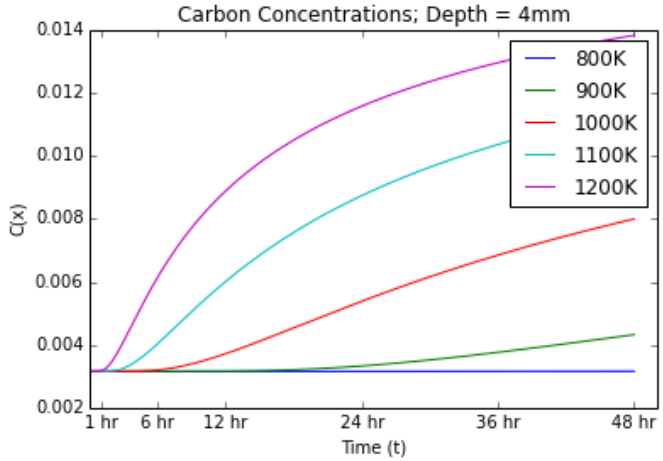


Figure 1 and 2 display the relationship between time, temperature and the concentration of carbon in depths of 2mm and 4mm respectively. The color pallet represents the concentration of carbon, the x axis is time and the y axis is temperature. These graphs prove that as temperature and time increase, so does the concentration.

Figure 3 and 4 display the carbon concentrations over time at different specified temperatures with a depth of 2mm and 4mm respectively. Here we can see that when a greater temperature is

introduced, the concentration's rate of increasing rises significantly. Over time is also appears that each temperature seems to plateau at a certain concentration of carbon. I also noticed that when the depth is 4mm it takes a longer period of time for the concentration to get to the same level as a 2mm depth carbon alloy. This makes sense as you are having to heat up a greater area of carbon to make the concentration increase. After 48 hours at 1200K, for instance, the concentration of carbon at 2mm is around 0.017, whereas at 4mm it is only reached 0.014.

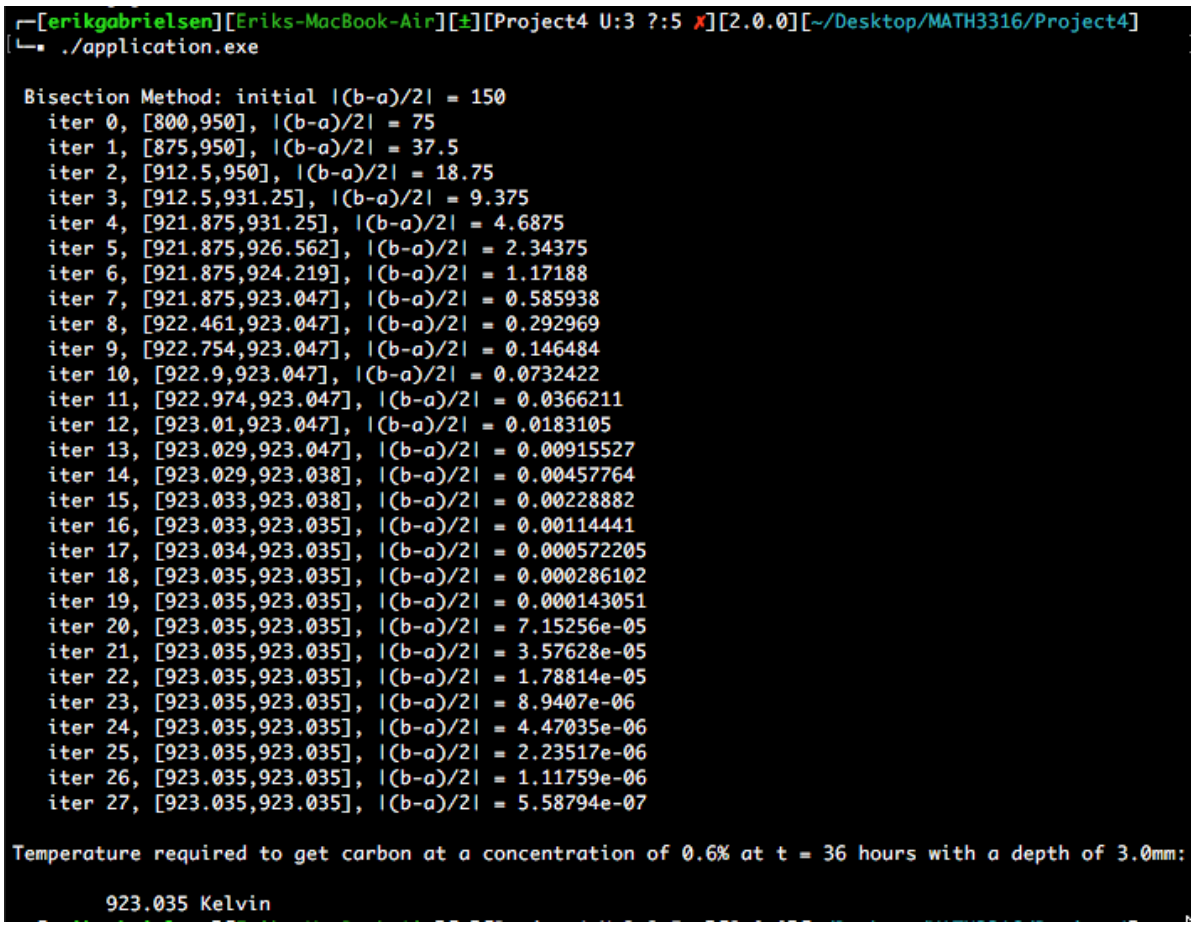
Code: *carbon.cpp*
test_carbon.cpp
carbon.ipynb

Part IV – Problem

In Part IV, I will use the carbon routine utilized in Part III to solve the following scenario:

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.

To do this I will use the bisection root finding function from Project 2 to find the correct temperature. I therefore need to set up the carbon equation where I have all variables on one side and 0 on the other. I need to have my root finding routine solve for $C(x, t, T) – 0.006 = 0$. Based on the graphs in part III I chose to start bisection at the interval [800, 1100] because I know that the temperature must be less then the 4mm depth steel at t = 36 hours and c(x) = 0.6% (1100K) and greater than at 2mm (800K). My results are on the following page:



Based on these results I was correct in my interval guess as the answer is approximately 923.035 Kelvin which is within my range of 800 – 1100 Kelvin. I also met the requirement that the error must be within 0.0001K as my error is 5.58x10-7, well below the required error. If I had chosen a better interval, such as [900, 950] bisection would have used less intervals to converge to a correct value of T. However, with what I was given I guessed pretty well.

Code: *application.cpp*
bisection.cpp

Appendix A – Source Code

application.cpp

```cpp
/* Erik Gabrielsen
   MATH 3316
   30 November 2015 */

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

using namespace std;

const double atolerance = 1e-15;
const double rtolerance = 1e-14;
const double depth = .003;
const double t = 129600; // 36 hours

double bisection(Fcn& f, double a, double b, int maxit, double tol, bool show_iterates);
double carbon(const double x, const double t, const double T, const double rtol, const double atol);

class fc : public Fcn {
public:
  double operator()(double T) {
    return carbon(depth, t, T, rtolerance, atolerance) - 0.006;
  }
};

int main() {
  fc f;

  // using bisection to solve for the roots of the above function.
  // I chose the interval [800, 1100]
  double temperature = bisection(f, 800, 1100, 100, 1e-6, true);

  cout << "\nTemperature required to get carbon at a concentration of 0.006 at t = 36 hours with a
    depth of 3.0mm: " << endl;
  cout << "\t" << temperature << " Kelvin" << endl;

  return 0;
}
```

bisection.cpp

```cpp
/* Erik Gabrielsen
   Code written by 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
```

carbon.cpp

```cpp
/* Erik Gabrielsen
   Math 3316
   27 November 2015 */

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

using namespace std;

const double PI = atan(1.0)*4;

// including adaptive_int
int adaptive_int(Fcn& f, const double a, const double b, const double rtol, const double atol, double& R, int&

class fcn : public Fcn {
public:
  double operator()(double z) {   // function evaluation of e^((-z)^2)
    return exp(-(z*z));
  }
};

// -- initial values of C0
const double C0 = .001;
const double Cs = .02;

double erf(const double y, const double rtol, const double atol) {
  fcn f; // can I instead pass this as an argument to erf?
  double R;
  int n, nTot;

  if (adaptive_int(f, 0, y, rtol, atol, R, n, nTot) == 1) {
    R *= 2.0/sqrt(PI);
  }
  return R;
}

double D(const double T) {
  return 6.2e-7*exp(-8e4/(8.31*T));
}

double carbon(const double x, const double t, const double T, const double rtol, const double atol) {
  return Cs - (Cs-C0)* erf((x/sqrt(4*t*D(T))), rtol, atol);
}
```

composite_int.cpp

```cpp
/* Erik Gabrielsen
    MATH 3316
    Project 4 */

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


using namespace std;

// -- optimal evaluation points for 4 node Gaussian numerical integration
const double x1 = -0.8611363116; //to avoid calling the sqrt function
const double x2 = -0.3399810436;
const double x3 = 0.3399810436;
const double x4 = 0.8611363116;
const double w1 = 0.3478548451;
const double w2 = 0.6521451549;
const double w3 = 0.6521451549;
const double w4 = 0.3478548451;

double composite_int(Fcn& f, const double a, const double b, const int n) {
  double h = (b - a) / n;

  double total = 0;

  double xmid, node_1, node_2, node_3, node_4;

  // calculating subintervals
  for (int i=0; i < n; i++) {
    xmid = a + (i+0.5)*h;
    node_1 = xmid + 0.5 * h * x1;
    node_2 = xmid + 0.5 * h * x2;
    node_3 = xmid + 0.5 * h * x3;
    node_4 = xmid + 0.5 * h * x4;

    total += 0.5 * h * (w1* f(node_1) + w2 * f(node_2) + w3 * f(node_3) + w4 * f(node_4));
  }
  return total;
}
```

Makefile

```
#####################################################################
#  Makefile for project 4
#
#  Erik Gabrielsen
#  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 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 $@

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

application.exe: application.cpp carbon.cpp bisection.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 #######
```

test_adapt.cpp

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

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


using namespace std;

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

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 main() {
    // limits of integration
    fcn f;
    const double a = -1;
    const double b = 7;
    f.c = 0.5;
    f.d = 25.0;


    // actual integral value
    double trueVal = 1.0 / f.c * (exp(f.c*b) - exp(f.c*a)) - 1.0 / f.d * (cos(f.d*b) - cos(f.d*a));
    printf("True integral value = %22.16e\n\n", trueVal);

    // test values
    double rtols[] = { 1e-2, 1e-4, 1e-6, 1e-8, 1e-10, 1e-12 };
    double atols[] = { 1e-5, 1e-7, 1e-9, 1e-11, 1e-13, 1e-15 };

    std::cout << "    rtol     atol      n    nTot        R         |I(f)-R(f)|  rtol|I(f)|+atol     Passed" << std::endl;
    for(int i = 0; i < 6; i++) { // for each pair of tols
        double R;
        int n, nTot = 0;
        int passed = adaptive_int(f, a, b, rtols[i], atols[i], R, n, nTot);
        double err = fabs(trueVal - R);
        double errBound = rtols[i] * fabs(trueVal) + atols[i];
        if (passed == 0) {
            printf("  %2.1e | %2.1e | %3u | %5u | %12.6e |  %7.1e  |   %7.1e      |      yes\n", rtols[i], atols[i], n, nTot, R, err, errBound);
        } else {
          printf("  %2.1e | %2.1e | %3u | %5u | %12.6e |  %7.1e  |   %7.1e      |       no\n", rtols[i], atols[i], n, nTot, R, err, errBound);
        }

    }
    cout << endl;
}
```

test_carbon.cpp

```cpp
/* Erik Gabrielsen
   Math 3316
   27 November 2015 */

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

using namespace std;

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

const double atolerance = 1e-11;
const double rtolerance = 1e-15;

int main() {
  Matrix K = Linspace(800, 1200, 400); // Kelvin
  Matrix t = Linspace(1, 172800, 600); // 48 hours (172800 s)
  Matrix c2mm = Matrix(400, 600);
  Matrix c4mm = Matrix(400, 600);
  Matrix C2mm_800K = Matrix(600);
  Matrix C2mm_900K = Matrix(600);
  Matrix C2mm_1000K = Matrix(600);
  Matrix C2mm_1100K = Matrix(600);
  Matrix C2mm_1200K = Matrix(600);
  Matrix C4mm_800K = Matrix(600);
  Matrix C4mm_900K = Matrix(600);
  Matrix C4mm_1000K = Matrix(600);
  Matrix C4mm_1100K = Matrix(600);
  Matrix C4mm_1200K = Matrix(600);

  for (int i = 0; i < 400; i++) {
    for (int j = 0; j< 600; j++) {
      c2mm(i, j) = carbon(.002, t(j), K(i), rtolerance, atolerance);
      c4mm(i, j) = carbon(.004, t(j), K(i), rtolerance, atolerance);
    }
  }
  for (int i = 0; i < 600; i++) {
    C2mm_800K(i) = carbon(.002, t(i), 800, rtolerance, atolerance);
    C2mm_900K(i) = carbon(.002, t(i), 900, rtolerance, atolerance);
    C2mm_1000K(i) = carbon(.002, t(i), 1000, rtolerance, atolerance);
    C2mm_1100K(i) = carbon(.002, t(i), 1100, rtolerance, atolerance);
    C2mm_1200K(i) = carbon(.002, t(i), 1200, rtolerance, atolerance);
    C4mm_800K(i) = carbon(.004, t(i), 800, rtolerance, atolerance);
    C4mm_900K(i) = carbon(.004, t(i), 900, rtolerance, atolerance);
    C4mm_1000K(i) = carbon(.004, t(i), 1000, rtolerance, atolerance);
    C4mm_1100K(i) = carbon(.004, t(i), 1100, rtolerance, atolerance);
    C4mm_1200K(i) = carbon(.004, t(i), 1200, rtolerance, atolerance);
  }

  K.Write("Temp.txt");
  t.Write("time.txt");
  c2mm.Write("c2mm.txt");
  c4mm.Write("c4mm.txt");
  C2mm_800K.Write("C2mm_800K.txt");
  C2mm_900K.Write("C2mm_900K.txt");
  C2mm_1000K.Write("C2mm_1000K.txt");
  C2mm_1100K.Write("C2mm_1100K.txt");
  C2mm_1200K.Write("C2mm_1200K.txt");
  C4mm_800K.Write("C4mm_800K.txt");
  C4mm_900K.Write("C4mm_900K.txt");
  C4mm_1000K.Write("C4mm_1000K.txt");
  C4mm_1100K.Write("C4mm_1100K.txt");
  C4mm_1200K.Write("C4mm_1200K.txt");
  return 0;
}
```

test_int.cpp

```cpp
/* Erik Gabrielsen
   SMU Mathematics
   Math 3316
   31 October 2015 */

// 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, const int 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 composite_int 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 Comp Int approximation:\n";
  cout << "     n             R(f)            relerr    conv rate\n";
  cout << "  ------------------------------------------------\n";
  vector<int> n = {20, 40, 60, 80, 100, 120, 140, 160, 180, 200};
  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";
}
```