Project4_Report

Will Spurgin

December 3, 2013

Part I

Report over Project 4: Numerical Integration

0.1 Overview

In this Project, I created a Composite Integration method the was at least $\mathcal{O}(h^6)$ accurate. The particular method I used is known as the Gauss-3 method. From there I created and Adaptive Integration function that used my Gauss-3 method of integration to integrate a given function to a given accuracy. In the last portions of the project I used my adaptive integration function to calculate carbon concentration in steel during the process of "carburizing" as a function of time and distance. In the last part of the lab, I used root finding algorithms from previous labs and solved for the time it takes for a carbon concentration of a metal to get to a specific percentage at a specified distance.

0.2 Part 1: High-order Numerical Integration

This portion of the lab, I was tasked with creating a Numerical Integration method that was at least $\mathcal{O}(h^6)$ accurate. Below is the code for both composite_int.cpp and test_int.cpp respectively.

```
//composite_int.cpp

/*
Will Spurgin
11/24/2013
High Performance Scietific Computing
MATH 3316
*/

//inclusions
#include <iostream>

using namespace std;

/*
Composite Integration function will be a Gaussian quadrature
rule with 3 points per subinterval to achieve O(h^6) accuracy.

The function 'fun' below must have the following syntax: y = fun(x)
```

```
Usage, F = composite_int(fun, a, b, n)

Where
    fun = integrand
    a = lower limit of integration
    b = upper limit of integration
    n = number of subintervals

    F = value of numerical integral
*/
double composite_int(double (*f)(const double), const double a, const
double b,
    const int n)
{
    if (b < a)
    {
        cerr << "error: illegal interval, b < a" << endl;
        return 0.0;
    }
    if (n < 1) {
        cerr << "error: illegal number of subintervals, n < 1" <<
endl;
        return 0.0;
    }

    //subinterval width
    double h = (b-a)/n;

    //set weights and nodes
    double x1 = -.774596669241483; //to avoid calling the sqrt
function
    double x2 = 0.0;
    double x3 = .774596669241483;
    double w1 = 5.0/9.0;
    double w2 = 8.0/9.0;
    double w3 = 5.0/9.0;

    double F = 0.0;

    double xmid, node1, node2, node3;
    //Iterate through subintervals
    for(int i = 0; i < n; i++)
    {

        // find evaluations points
        xmid  = a + (i+0.5)*h;
        node1 = xmid + 0.5*h*x1;
        node2 = xmid + 0.5*h*x2;
        node3 = xmid + 0.5*h*x3;

        // add approximation to final result
        F += 0.5*h*(w1*f(node1) + w2*f(node2) + w3*f(node3));
    }
```

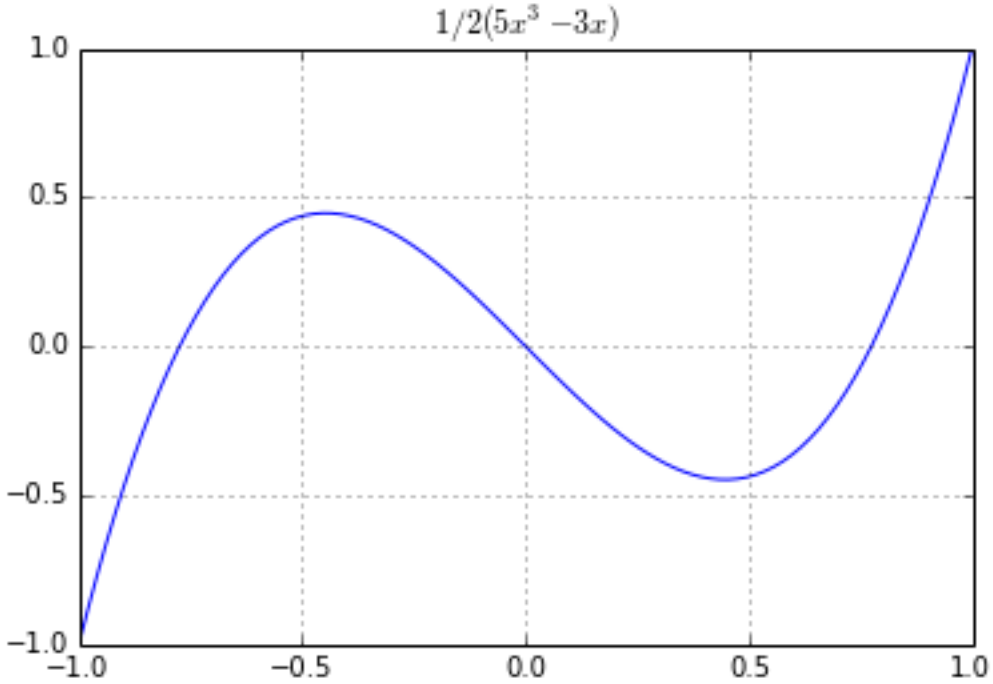
```
    return F;
}

//END OF FILE

//test_int.cpp

/*
Will Spurgin
11/20/2013
High Performance Scientific Computing
MATH 3316
*/

/*
The following is an adaptation of Daniel R. Reynolds
'test_Gauss2.cpp'.
The majority of the code written below is his with only slight
modifications
in order to test the 'composite_int' function. Let credit be given
where
credit is due.
*/

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

using namespace std;

// function prototypes
double composite_int(double (*f)(const double), const double a, const
double b,
  const int n);

// Integrand
const double c=3.0;
const double d=20.0;
double f(const double x) {
  return (exp(c*x) + sin(d*x));
}


// This routine tests the Gauss-2 method on a simple integral
int main(int argc, char* argv[]) {

  // limits of integration
  double a = -5.0;
  double b = 2.0;

  // true integral value
```

```
  double Itrue = 1.0/c*(exp(c*b) - exp(c*a)) -
1.0/d*(cos(d*b)-cos(d*a));
  printf("\n True I = %22.16e\n", Itrue);


  // test the composite_int which is Gauss-3 rule
  cout << "\n Composite-int (Gauss-3) rule:\n";
  cout << "     n              R              relerr    conv rate\n";
  cout << "  ---------------------------------------------------\n";
  int n[] = { 10, 20, 40, 80, 160, 320, 640, 1280, 2560, 5120 };
  int ntests=10;

  // iterate over n values, computing approximations, error,
convergence rate
  double Iapprox, olderr, relerr=0.0;
  for (int i=0; i < ntests; i++) {

    printf("   %6i", n[i]);
    olderr = relerr;
    Iapprox = composite_int(f, a, b, n[i]);
    relerr = fabs(Itrue-Iapprox)/fabs(Itrue);
    if (i == 0) {
      printf("  %22.16e  %7.1e\n", Iapprox, relerr);
    } else {
      printf("  %22.16e  %7.1e   %f\n", Iapprox, relerr,
         (log(olderr) - log(relerr))/(log(1.0/n[i-1]) -
log(1.0/n[i])));
    }

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

}

//END OF FILE
```

The composite integration method I choose was the Gauss-3 method. This method was fairly easy to implement because the Gauss-2 method had already been provided. The Gauss-3 included only one more node, weight, and evaluation point. The optimum evaluation points, which shown above are $\pm 0.774596669241483, 0$, which are also the root for the $P_3(x)$ Legendre Polynomial (shown below)

In [16]:
```
x = linspace(-1, 1, 100)
f = .5*(5*x**3-3*x)
figure()
plot(x, f)
grid(True)
title("$1/2(5x^3-3x) $")
```

Out [16]:
```
<matplotlib.text.Text at 0x1094617d0>
```



To find the roots of this polynomial, I used Newton's method for root finding that I build for Project 2. After doing that, creating the rest of the Gauss-3 method involved the same procedure as the Gauss-2 with the one more evaluation point, weight, and node.

Running the test_int main above, versus the test_Gauss2 revealed the following:

In [18]:
```
import os
out = os.popen("make test_int.out test_Gauss2.out").read()
print out
out = os.popen("./test_Gauss2.out").read()
print out
out = os.popen("./test_int.out").read()
print out
```

```
clang++ -O2 test_int.cpp composite_int.cpp -o test_int.out
clang++ -O2 test_Gauss2.cpp composite_Gauss2.cpp -o test_Gauss2.out


 True I = 1.3455272724230790e+02

 Gauss-2 rule:
     n              R              relerr    conv rate
  ---------------------------------------------------
       20  1.3477150984992036e+02  1.6e-03
       40  1.3454620482188116e+02  4.8e-05   5.067947
       80  1.3455239619543735e+02  2.5e-06   4.300300
      160  1.3455270737709154e+02  1.5e-07   4.058719
      320  1.3455272601262266e+02  9.1e-09   4.013884
      640  1.3455272716563465e+02  5.7e-10   4.003422
     1280  1.3455272723751864e+02  3.6e-11   4.000849
     2560  1.3455272724200844e+02  2.2e-12   3.999409
     5120  1.3455272724228905e+02  1.4e-13   3.990175
```

```
  10240  1.3455272724230659e+02  9.7e-15   3.849303
 ---------------------------------------------------


True I = 1.3455272724230790e+02

Composite-int (Gauss-3) rule:
    n              R              relerr    conv rate
 ---------------------------------------------------
      10  1.3512834428399557e+02  4.3e-03
      20  1.3452227370687439e+02  2.3e-04   4.240428
      40  1.3455283872388040e+02  8.3e-07   8.093661
      80  1.3455272843370011e+02  8.9e-09   6.548013
     160  1.3455272725944172e+02  1.3e-10   6.119657
     320  1.3455272724257014e+02  1.9e-12   6.029777
     640  1.3455272724231185e+02  2.9e-14   6.052705
    1280  1.3455272724230784e+02  4.2e-16   6.118941
    2560  1.3455272724230772e+02  1.3e-15   -1.584963
    5120  1.3455272724230781e+02  6.3e-16   1.000000
 ---------------------------------------------------
```

You can see that the difference between the Gauss-2 and Gauss-3 is substancial. The Gauss-3 rule converges around 1280 nodes while the Gauss-2 is still converging at 10240 nodes. Almost a factor of 10, with the only difference being the extra node, weights, and evaluation points. That extra term in the Gauss-3 rule (see code above) makes all the difference. In general a Gauss-M rule has accuracy $\mathcal{O}(h^{2M+2})$ Therefore the Gauss-2 rule has $\mathcal{O}(h^6)$ accuracy and the Gauss-3 has $\mathcal{O}(h^8)$. Therefore, the error, or the missing portion of accuracy for the Gauss-2 rule can be said to be $\mathcal{O}(h^7)$. The Gauss-3 rule is missing $\mathcal{O}(h^9)$ which is much smaller than $\mathcal{O}(h^7)$. So the Gauss-3 is effectively going to get "more right" every time. Look at the evaulation at $n = 20$ for both Gauss-2 and Gauss-3. The Gauss-3 has the first 4 digits right, where the Gauss-2 has only the first 3 correct.

0.3 Part 2: Adaptive Numerical Integration

The second part of this lab was to take the Composite Integration method constructed in part 1, and create an Adaptive Integration using it. Adaptive integration's goal is to create a method to compute the integral to a given accuracy, 10^{-10} for example. And do so without knowing what the true integral is. Showen below is the code for "adaptive_int.cpp" and "test_adapt.cpp" respectively.

```
//adaptive_int.cpp

/*
Will Spurgin
11/24/2013
High Performance Scientific Computing
MATH 3316
*/

#include <iostream>
#include <cmath>

using namespace std;

//prototype
double composite_int(double (*f)(const double), const double a, const
```

```
double b,
  const int n);

int adaptive_int(double (*f)(const double), const double a, const
double b,
    const double rtol, const double atol, double &R, int &n)
{

    //set up the interval iterator
    int iterator;
    if(rtol >= 1e-2)
        iterator = 3;
    else if (rtol >= 1e-4)
        iterator = 4;
    else if (rtol >= 1e-6)
        iterator = 10;
    else if (rtol >= 1e-8)
        iterator = 14;
    else
        iterator = 35;

    //primary read
    int maxit = 700;
    int intervals = 1;
    double previous, current;

    n = intervals;

    previous = composite_int(f, a, b, intervals);
    bool converged = false;

    for(int i = 0; i < maxit; i++)
    {
        intervals += iterator;
        current = composite_int(f, a, b, intervals);

        if(abs(current - previous) < (rtol*current + atol))
        {
            converged = true;
            break;
        }
        previous = current;
        n += intervals;
    }
    R = current;

    if(converged)
        return 0;
    else
        return 1;
}

// END OF FILE
```

```
// test_adapt.cpp

/*
Will Spurgin
11/24/2013
High Performance Scientific Computing
MATH 3316
*/

#include <iostream>
#include <cmath>

using namespace std;

//prototypes
int adaptive_int(double (*f)(const double), const double a, const
double b,
    const double rtol, const double atol, double &R, int &n);

const double c=3.0;
const double d=20.0;
double f(const double x) {
  return (exp(c*x) + sin(d*x));
}

int main(int argc, char** argv)
{
    double atol = 1e-15;
    double rtol[] = { 1e-2, 1e-4, 1e-6, 1e-8, 1e-10, 1e-12 };
    double a = -5.0;
    double b = 2.0;

    cout << "rtol               R              n        relerr" <<
endl
        << "-------------------------------------------------" <<
endl;
        double Rapprx;
        for(int i = 0; i < 6; i++)
        {
            double R = 0.0;
            int n = 0;
            if(adaptive_int(f, a, b, rtol[i], atol, R, n) == 0)
            {
                if(i != 0)
                {
                    double relerr = fabs(Rapprx-R)/fabs(Rapprx);
                    printf("%7.1e  %22.16e %i       %7.1e\n", rtol[i],
R, n, relerr);
                }
                else
                    printf("%7.1e  %22.16e %i\n", rtol[i], R, n);
            }
            else
            {
```

```
                printf("%7.1e  %22.16e  %i\n", rtol[i], R, n);
                cout << " (Did not converge) " << endl;
            }
            Rapprx = R;
        }
    return 0;
}

// END OF FILE
```

The Adaptive strategy is fairly straight forward. Without knowing the true integral, we can approximate

$$|I(f) - R_n(f)| < rtol|I(f)| + atol$$

with

$$|R_{n+k}(f) - R_n(f)| < rtol|R_{n+k}(f)| + atol$$

Where $k = \mathbb{R}^+$. For my method, I chose k adaptively based on the desired "rtol" (values seen above). The goal is to minimize the total amount of work the method has to do by calling the composite_int function with varing n values. For example if a method had $k = 1$ and let's assume that a certain accuracy A could be achieved at $n = 10$ for a certain integral $\int_a^b f(x)\ dx$ The method, starting at $n = 1$ would call the composite_int function all 10 times using a total count of nodes (which equates to how many times the loop in the composite_int function is run) at $\sum_{i=1}^{10} i = 55$ Where as my method we will say it chose $k = 3$ on the same problem, whould call the composite_int function 4 times and have a total count of nodes at $\sum_{i=0}^{3} 3i + 1 = 22$.

The output for the test_adapt is shown below:

In [61]:
```
out = os.popen("make test_adapt.out").read()
print out
out = os.popen("./test_adapt.out").read()
print out
```

```
clang++ -O2 test_adapt.cpp adaptive_int.cpp composite_int.cpp -o
test_adapt.out

rtol                R               n         relerr
-------------------------------------------------------
1.0e-02  1.3373072903912038e+02     5
1.0e-04  1.3451146824659810e+02    45      3.1e-04
1.0e-06  1.3455274836496304e+02   105      1.6e-07
1.0e-08  1.3455272756139320e+02   301      2.4e-09
1.0e-10  1.3455272724552859e+02   531      2.4e-11
1.0e-12  1.3455272724245839e+02  1585      1.1e-12
```

There are some trade offs with my method. Though I choose (as seen above) k adaptively, it is not perfect. There is no set functional relationship with what k should be and the tolerance. Through expiremention, I have chosen the values shown in "adaptive.cpp" bassed entirely on the difference between "n" and "relerr" shown above. It was my goal in my method to achieve the smallest amount for n without having to work too hard to do so. The "working too hard" is shown in the "relerr" which, in this case at least, is relatively close to the actual "rtol" without exceeding it by too much. I could have chosen $k = 500$ all the time and made the lower "rtols" at the end have lower values of "n" (lowest being 502 for this k), however the "relerr" could be far smaller than necessary, and certainly, the higher "rtols" in the beginning would have far greater "n" and smaller "relerr" than necessary.

It would be very interesting if there were a computable relation between the "rtol" and the function to be integrated that could always tell the perfect value for the "iterator", but that seems a bit like the wishful thinking to me.

0.4 Part 3: Application, Carbon Concentrations

In this portion of the project, I take the adaptive_int function from Part 2 and put it through some "application testing." The application is that of Carbon concentrations in steel during the process of "carburizing." This function that defines the concentration of Carbon in steel is

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

Where C_0 is the initial Carbon concentration of the steel, C_s is the Carbon concentration of a gas that is exposed to the steel, D is the diffusion constant of the steel at a set temperature (we ignore temperature in this problem), x is the distance of the gas from the steel's surface, and t is the elapsed time. Further the function $erf(y)$ is defined as,

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

Part 3 called for the creation of a "carbon" function that would calculate the Carbon concentration of a steel with varying x and t values ($C_s, C_0, \text{and } D$ were considered constant to simplify our problem), as well as a "test_carbon" main that would rigorously test the "carbon" function. The code for both "carbon.cpp" and "test_carbon.cpp" are shown below respectively:

```
// carbon.cpp

/*
Will Spurgin
11/24/2013
High Performance Scietific Computing
MATH 3316
*/

#include <iostream>
#include <cmath>
#ifndef PI
    #define PI 3.141592653589793
#endif

using namespace std;

//prototypes
int adaptive_int(double (*f)(const double), const double a, const
double b,
    const double rtol, const double atol, double &R, int &n);

double f_(const double x) { return (exp(-pow(x,2))); }

double erf(const double y, const double rtol, const double atol)
{
    double R; int n;
    int errors = adaptive_int(f_, 0, y, rtol, atol, R, n);
    if(!errors)
        return (2/sqrt(PI)*R);
    else
    {
        cerr << "The error did not converge with y = " << y << endl;
        return (2/sqrt(PI)*R);
```

```
    }
}

double carbon(const double x, const double t,const double rtol,
    const double atol)
{
    double C_s = 0.1;
    double C_0 = 0.001;
    double D = 5e-11;
    double denom = (sqrt(4*D*t));
    if(denom == 0)
        return 0.0;
    double z = (x/denom);
    return (C_s - (C_s - C_0)*erf(z, rtol, atol));
}

// END OF FILE

// test_carbon.cpp

/*
Will Spurgin
11/24/2013
High Performance Scietific Computing
MATH 3316
*/

#include "mat.h"
#include <string>
#include <iostream>

using namespace std;

//prototypes
double carbon(const double x, const double t,const double rtol,
    const double atol);

int main(int argc, char** argv)
{
    Mat x_ = Linspace(0, 4e-3, 400);
    x_.Write("x.txt");
    double* x = x_.get_data();

    Mat t_ = Linspace(0, 172800, 800);
    t_.Write("t.txt");
    double* t = t_.get_data();

    double rtol = 1e-10;
    double atol = 1e-15;

    Mat C(400, 800);
    for(int i = 0; i < 400; i++)
    {
        for(int j = 0; j < 800; j++)
```

```
        {
            C(i, j) = carbon(x[i], t[j], rtol, atol);
        }
    }
    C.Write("C.txt");
    cout << "Wrote initial output for C(x, t). Moving to hourly
calculations"
         << endl;

    Mat C_hours(400, 1);
    double seconds[] = { 3600, 21600, 43200, 86400, 172800 };
    string output_files[] = { "C_1hour.txt", "C_6hour.txt",
"C_12hour.txt",
        "C_24hour.txt", "C_48hour.txt" };
    for(int i = 0; i < 5; i++)
    {
        for(int j = 0; j < 400; j++)
        {
            C_hours(j) = carbon(x[j], seconds[i], rtol, atol);
        }
        C_hours.Write(output_files[i].c_str());
    }

    Mat C_3(800, 1);
    for(int i = 0; i < 800; i++)
        C_3(i) = carbon(3e-3, t[i], rtol, atol);
    C_3.Write("C_3mm.txt");

    cout << "Done." << endl;

    return 0;
}

// END OF FILE
```

This Part of the project calls for the creation of several input arrays "Create an array of 400 evenly-spaced x values over the interval [0; 4] mm .... Create an array of 800 evenly-spaced t values over the interval [0; 48] hours", but it should be noted that the values in the "Linspace" Mat function I have for creating the x and t values are neither 4 nor 48 for either respectively. I made the choice to convert from milimeters for the x interval to meters, and the hours for the t interval to seconds. The main reason I have for doing this is the fact that for the formula stated above for computing $C(x,t)$, the units matter (i.e. 1 hour $\neq$ 1 second) If we take just the portion inside the $erf(y)$ part for $C(x,t)$ we know y's units follow as thus:

$$\frac{m}{\sqrt{\frac{m^2}{s}s}} = \frac{m}{\frac{m}{s}} = s$$

Where m is meters and s is seconds. We know this because of the units of D and that logically, the function $erf(y)$ is a function of time.

Now imagine taking $erf\left(\frac{x}{\sqrt{4Dt}}\right)$ and using first $t = 1$ but instead of t being seconds, having t be in hours. Computationally, with $D = 5 \times 10^{-11}$ and say $x = 1$ meter, inside the sqaure root we have,

$$4 \cdot (5 \times 10^{-11}) \cdot 1$$

Which is 2×10^{-10} In fact, this is the wrong value for what is actually supposed to be inside the square root. Or is it? It is. The technically speaking the units for 2×10^{-10} are $\frac{m^2 hrs}{s}$ which if units convert there quantities

magically would be totally fine for the remainder of the computation. However they don't (yet). The actual quantity is $4 \cdot (5 \times 10^{-11}) \cdot 1(3600)$ while the value inside the square root between these two different figures is only a few extra decimal places away from zero, when computing the fraction those decimal places translate to thousands if not hundreds of thousands of whole numbers apart.

Thus I chose to convert the intervals to their proper units before computing them. The output for test_carbon.cpp is shown below as well as the graphs from "carbon.ipynb":

In [72]:
```
out = os.popen("make test_carbon.out").read()
print out
out = os.popen("./test_carbon.out").read()
print out
out = os.popen(" ipython nbconvert --to python carbon.ipynb").read()
print out
%loadpy carbon.py
```

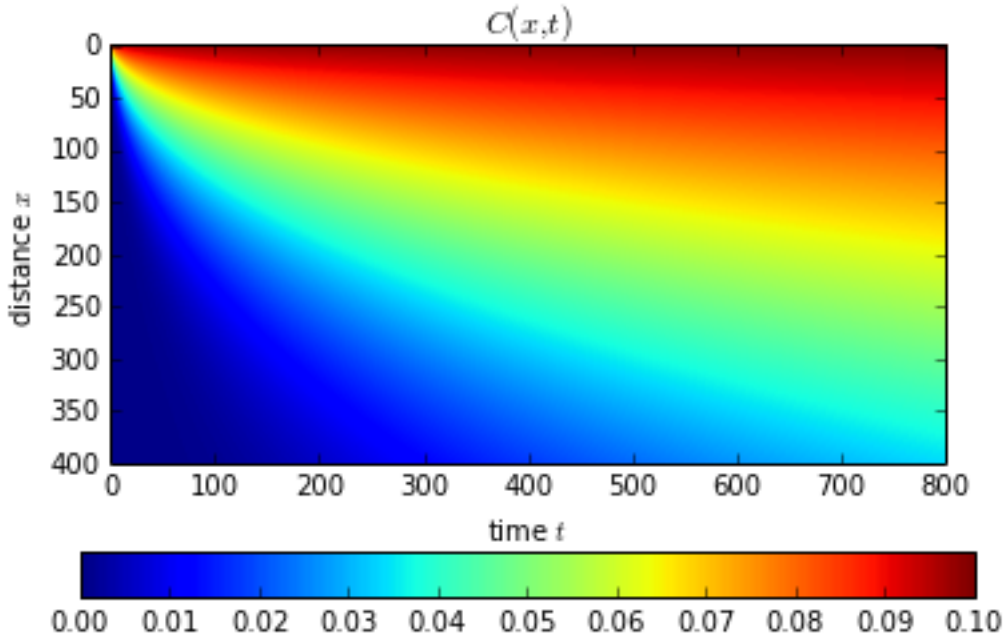
```
make: 'test_carbon.out' is up to date.

Wrote initial output for C(x, t). Moving to hourly calculations
Done.
```

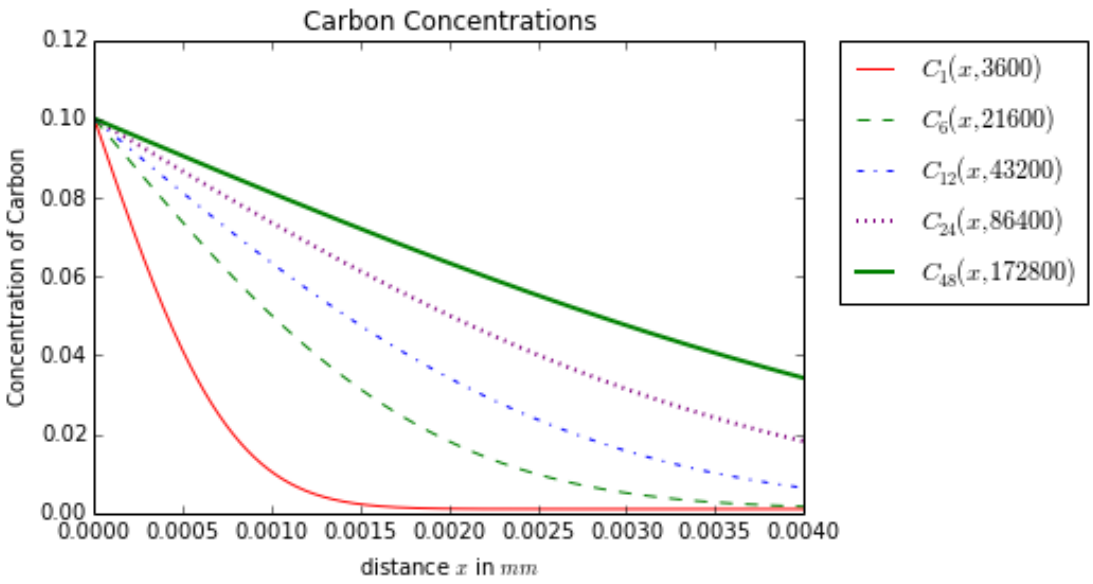
In [74]:
```
# In[7]:

x = loadtxt("x.txt")
t = loadtxt("t.txt")
C = loadtxt("C.txt")
C_1 = loadtxt("C_1hour.txt")
C_6 = loadtxt("C_6hour.txt")
C_12 = loadtxt("C_12hour.txt")
C_24 = loadtxt("C_24hour.txt")
C_48 = loadtxt("C_48hour.txt")


# In[10]:

figure()
imshow(C)
colorbar(orientation='horizontal')
xlabel("time $t$")
ylabel("distance $x$")
title("$C(x,t)$")


# Out[10]:

#     <matplotlib.text.Text at 0x109b3ca50>

# image file:

# In[33]:

figure()
plot(x, C_1, 'r', label="$C_{1}(x, 3600)$")
plot(x, C_6, 'g--', label="$C_{6}(x, 21600)$")
plot(x, C_12, 'b-.', label="$C_{12}(x, 43200)$")
plot(x, C_24, color="purple", lw=2, ls=":", label="$C_{24}(x, 86400)$")
plot(x, C_48, color="green", lw=2, ls="-", label="$C_{48}(x, 172800)$")
xlabel("distance $x$ in $mm$")
ylabel("Concentration of Carbon")
title("Carbon Concentrations")
legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
```

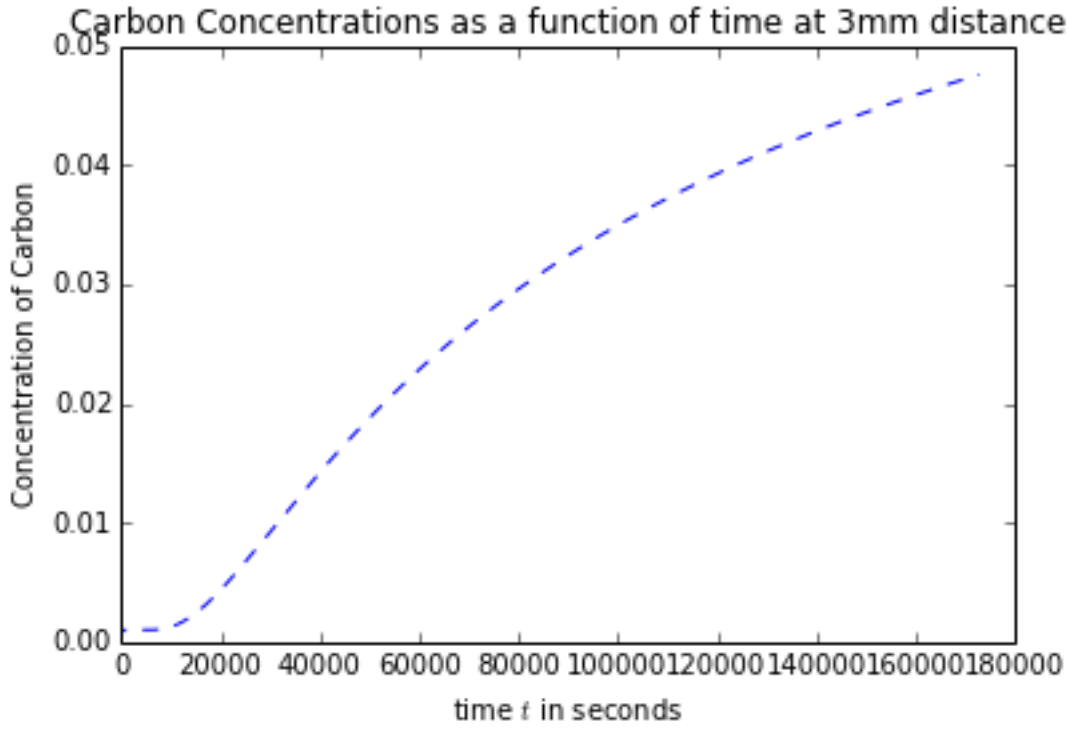
```
# Out[33]:

#     <matplotlib.legend.Legend at 0x10d9715d0>

# image file:

# In[30]:

C_3mm = loadtxt("C_3mm.txt")
figure()
plot(t, C_3mm, 'b--')
xlabel("time $t$ in seconds")
ylabel("Concentration of Carbon")
title("Carbon Concentrations as a function of time at 3mm distance")


# Out[30]:

#     <matplotlib.text.Text at 0x10e713990>

# image file:

# In[ ]:
```

Out [74]:

```
<matplotlib.text.Text at 0x10c43a990>
```







The contour plot (plot 1) shows the distribution of values in the matrix "C" which has 400 rows of evenly spaced x values $[0; 4]$mm and 800 columns of evenly spaced t values $[0; 48]$ hours. The different shades of colors, which correspond to numerical values, represent the function values of $C(x, t)$. By this first plot, we see how the increase in distance greatly affects the time it takes for the Carbon concentration to increase.

The second plot shows that relationship further with a several plots of set t values of $[3600, 21600, 43200, 86400, 172800]$. Each line on plot 2 enforces the relationship established in plot 1. Take for instance the line $C_1(x, 3600)$ which shows Carbon concentrations at one hour for varying distances, when the distance is closer to zero, the higher the Carbon concentration is at an hour's time. Plot 2 also shows how the process

of diffusion between the gas and the steel is faster in the first few hours and begins to flatten, so to say, as the amount of time increases.

Plot 3 is an additional plot that allows for further understanding of the relationship between time, distance and the Carbon concentration. For this particular plot, I chose to make $x = 0.003$m. If we let $C(0.003, t) = f(t)$, then we see that $f'(t)$ begins to flatten at $t \approx 40000$ after an inital climb starting near $t = 20000$. This further supports the relationship presented in plot 2.

0.5 Part 4: Problem, Root Finding and Carbon Concentrations

The final part of the project asked to find the amount of time it would take a piece of steel to reach a Carbon concentration of 4% at 3mm distance with all the same with all the same constants from Part 3. This part required me to use one of my root finding algorithms from Project 2. The one I choose to use was the Foward Newton (or Forward Finite Newton) method. Below is the code for "application.cpp":

```
// application.cpp

/*
Will Spurgin
11/29/2013
High Performance Scientific Computing
MATH 3316
*/

#include <iostream>
#include <cmath>

using namespace std;

//function prototypes

//main carbon concentration function
double carbon(const double x, const double t,const double rtol,
    const double atol);

//Single variable function
double f(const double t);

//root finding function
double fd_newton(double (*f)(const double), double x,
        int maxit, double tol, double alpha);



int main(int argc, char** argv)
{
    double alpha = pow(2.0, -26);
    int maxit = 100;
    double tol = 1e-6;
    cout << "Calculating the time it takes for the carbon
concentration"
        << ", which initially is 0.1\% to get " << endl << "to 4\%
through "
        << "carburizing using a gas with a carbon concentration"
```

```
        << " of 10\% at a distance of 3mm from the metal" << endl;
    double t = fd_newton(f, 120000, maxit, tol, alpha);
    cout << "Approximated time for .04 = C(3e-3,t) or root finding
problem "
        << "0 = C(3e-3,t) - .04 is:" << endl << "t = ";
        printf("%6fs\n", t);
    return 0;
}

double f(const double t)
{
    return (carbon(3e-3, t, 1e-14, 1e-15) - .04);
}

// END OF FILE
```

The problem calls for $0.04 = C(0.003, t)$ solving for t. However to make this a root finding problem that is solvable for a root finding algorithm we have to have the problem be of the form $0 = f(x)$. Thankfully that with some quick algebra the problem becomes $0 = C(0.003, t) - 0.04$. Therefore I created the function "f" above which took one variable, 't', for time, had the distance of 0.003, relative tolerance 10^-14, and an absolute tolerance of 10^-16 for the original "carbon" function as well as the -0.04 to find the time it takes for a Carbon concentration of 4% to be achieved. Based on plot 3 from Part 3, we can see that at 3mm distance near 120000 seconds, the concentration is close to 4%, so I chose 120000 to be my initial guess for the Forward Finite Newton's method. Below are the results from "application.cpp"

In [78]:
```
out = os.popen("make application.out").read()
print out
out = os.popen("./application.out").read()
print out
```

```
clang++ -O2 application.cpp mat.cpp adaptive_int.cpp composite_int.cpp
carbon.cpp fd_newton.cpp -o application.out

Calculating the time it takes for the carbon concentration, which
initially is 0.1% to get
to 4% through carburizing using a gas with a carbon concentration of
10% at a distance of 3mm from the metal
 iter: 0 | x = 120000 | c = 123814 | absolute value of f(x) = 120000 |
err = 3814.12
 iter: 1 | x = 123814 | c = 123840 | absolute value of f(x) = 123814 |
err = 25.3905
 iter: 2 | x = 123840 | c = 123839 | absolute value of f(x) = 123840 |
err = 0.126303
 iter: 3 | x = 123839 | c = 123839 | absolute value of f(x) = 123839 |
err = 8.6051e-06
 iter: 4 | x = 123839 | c = 123839 | absolute value of f(x) = 123839 |
err = 6.27479e-08
Approximated time for .04 = C(3e-3,t) or root finding problem 0 =
C(3e-3,t) - .04 is:
t = 123839.386989s
```

According to my results:

$$C(0.003, 123839.386989) = 0.04$$

The tolerance I gave the "fd_newton" function was 10^{-6} which would ensure that my answer was correct to the micro-second. Looking at the graph from Part 3, this result seems logical.

0.6 Conclusion

This Project was designed to create an efficient not just accurate Numerical Integration method and Adaptive strategy. I believe, while my methods could have been more efficient, that they were definitely far better than other techniques. I tested this by using a Adaptive strategy I mentioned above, incrementing by 1 for intevals. With that method, it took nearly 10 minutes to complete the task in "test_carbon.cpp". My strategy completes the task in under a second. Both accomplish the task, but I certainly would not want to wait around forever for the "plus one" strategy!Fin