Math 3316, Fall 2016
Due Dec. 1, 2016

Project 4 – Numerical Integration
The first two sections of this project will be checked in lab either Wednesday Nov. 16 or Monday Nov. 21 – this completion grade will count for 10% of the overall project. The final project is due by 5:00 pm on Thursday, December 1, and should be uploaded to Canvas. Instructions on what should be turned in are included at the end of this document.

Late work will lose points based on the following schedule:

```
1 minute to 24 hours  10 points
24 hours to 48 hours  20 points
48 to 72 hours  30 points
72 to 96 hours  40 points
over 96 hours  no credit
```

1. High-order numerical integration:

A C++ function `composite_gauss2()` for approximating

$$\int_a^b f(x)\,\mathrm{d}x$$

using the composite Gaussian numerical integration formula with 2 nodes on each of n subintervals of $[a,b]$ has been provided on the course web page in the file `composite_Gauss2.cpp`. A C++ `main()` routine in the file `test_Gauss2.cpp` has also been provided to examine the $\mathcal{O}(h^4)$ convergence of the method.

For the first part of this project, create your own composite numerical integration function,

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

in the file `composite_int.cpp`. This function should implement a composite numerical integration formula that is $\mathcal{O}(h^8)$ accurate – this may be a method of your own invention, or a sufficiently-accurate method from the book.

You should then create a C++ `main()` routine in the file `test_int.cpp`, that emulates the provided routine `test_Gauss2.cpp` to ensure that your integration routine works as designed. This should use the same integrand as in the supplied routine, but you should modify the values of n used to better demonstrate the convergence rate of the method (use at least 6 n values). Include this output in your report.

Note: you will be calling this function numerous times throughout the remainder of the project, so one component of your grade on this part of the project will be the efficiency of your approach (mathematical method + algorithmic implementation).

2. Adaptive Numerical Integration:

A fundamental request of any solver is to provide a result of a desired *accuracy*, as opposed to just a result from a desired amount of work. In this part of the project, you will create an *adaptive* numerical integration function,

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

in a new file `adaptive_int.cpp` that will compute

$$\int_a^b f(x)\,\mathrm{d}x$$

by calling your `composite_int()` function in an adaptive manner. The goal here is to construct an approximation of the true integral such that

$$|I(f) - R_n(f)| < rtol\ |I(f)| + atol, \tag{1}$$

where

$$I(f) = \int_a^b f(x)\,\mathrm{d}x$$

and $R_n(f) =$ `composite_int(f, a, b, n)`, such that you achieve this result in the least possible computational effort (i.e. minimize n, but don't work too hard to do so).

The challenge with this error bound is that we do not know the true integral, $I(f)$. However, for rapidly-convergent methods (like yours), an approximation using more subintervals, e.g. $R_{n+k}(f)$ for $k \gtrsim 4$, will be more accurate than $R_n(f)$. Hence the equation (**??**) could be modeled with the approximation

$$|R_{n+k}(f) - R_n(f)| < rtol\ |R_{n+k}(f)| + atol. \tag{2}$$

Note: the "4" above was arbitrary, decide on a strategy of your own. Your `adaptive_int()` function should update n adaptively until this bound is satisfied. Your goal here is to obtain errors below the target tolerance, but not too far below, so that your solver does not work harder than necessary. At one extreme, updating `n+=1` will result in the minimum n, but could take too much work before an acceptable n is found; at the other extreme updating `n*=100` will rapidly find an acceptable n, but will almost certainly result in too much work. Be creative here.

Upon completion, `adaptive_int()` should fill `R` with its double-precision approximation of the integral, it should fill `n` with the final number of intervals it used in computing `R` (i.e. the number corresponding to your value of `R`), it should fill `Ntot` with the *total* number of intervals used along the way (i.e. including all failed attempts with an incorrect n), and it's return value should be 0 if it believes that it successfully satisfied your approximation of the error bound (**??**) or 1 if it failed at it's task. Create a C++ `main()` routine in the file `test_adapt.cpp` that uses your `adaptive_int()` function to integrate the same problem as before, but now using the pairs of tolerances $(rtol, atol)$ with $atol_i = rtol_i/1000$, and $rtol = \left\{10^{-2}, 10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}\right\}$. For each set of tolerances, print the actual values of $|I(f) - R(f)|$ and $rtol\,|I(f)| + atol$, as well as the `n` and `Ntot` required by your method. Does your method achieve the desired relative error? How hard did it have to work to do so? Explain your findings.

Note: you will also be calling this function numerous times throughout the remainder of the project, so a significant component of your grade on this part of the project will be the overall efficiency of your adaptivity strategy.

3. Application:

Steel can be hardened by increasing the concentration of carbon relative to iron in the alloy. This may be accomplished through a process named "carburizing," in which the steel is heated and exposed to a gas with high carbon concentration, causing the carbon to diffuse into the metal. At a constant temperature, the concentration of carbon $C(x,t,T)$ at a distance x (in meters) from the surface, at time t (in seconds), for a specified temperature T (in Kelvin) is given by the formula

$$C(x,t,T) = C_s - (C_s - C_0)\,\mathrm{erf}\left(\frac{x}{\sqrt{4t\,D(T)}}\right), \tag{3}$$

where C_0 is the initial carbon concentration in the steel, C_s is the carbon concentration in the gas, and $D(T)$ is the temperature-dependent diffusion coefficient of the steel,

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

The so-called *error function* is defined as

$$\mathrm{erf}(y) = \frac{2}{\sqrt{\pi}}\int_0^y e^{-z^2}\,\mathrm{d}z. \tag{5}$$

In this project, we will consider a steel with initial carbon concentration $C_0 = 0.001$ and gas carbon concentration $C_s = 0.02$.

Create a file named `carbon.cpp` that contains (at least) 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 of these functions should use your `adaptive_int()` function to evaluate equation (**??**). The second of these functions should use your `erf()` function to evaluate equation (**??**) for specified input values of x, t and T.

Write a C++ `main()` routine in a file `test_carbon.cpp` that performs the following tasks:

- Create an array of 400 evenly-spaced T values over the interval $[800, 1200]$ K. Output this to disk as the file `Temp.txt`.
- Create an array of 600 evenly-spaced t values from $t = 1$ second up to $t = 48$ hours. Output this to disk as the file `time.txt`.
- Create a 400×600 array that contains $C(0.002, t, T)$. Output this to disk as the file `C2mm.txt`.
- Create a 400×600 array that contains $C(0.004, t, T)$. Output this to disk as the file `C4mm.txt`.
- Create an array of length 600 containing $C(0.002, t, 800)$. Output to disk as the file `C2mm_800K.txt`. Repeat 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`).

- Create an array of length 600 containing $C(0.004, t, 800)$. Output to disk as the file C4mm_800K.txt. 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 should be computed using the tolerances $rtol = 10^{-11}$ and $atol = 10^{-15}$.

In an Jupyter notebook carbon.ipynb, load the above data files, and create the following plots:

- Create a filled contour plot of $C(0.002, t, T)$ with the commands

  ```
  figure()
  imshow(C2mm)
  colorbar(orientation='horizontal')
  ```

  (this assumes that you have stored the 2D array as "C2mm" in Python)

- Create a second figure with the filled contour plot of $C(0.004, t, T)$.

- Create 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 (i.e. 5 line plots overlaid in different colors, containing your data from the files C2mm_800K.txt, etc.).

  Use the command

  ```
  xticks( (1*3600, 6*3600, 12*3600, 24*3600, 36*3600, 48*3600),
          ('1 hr', '6 hr', '12 hr', '24 hr', '36 hr', '48 hr') )
  ```

  to label 1, 6, 12, 24, 36 and 48 hours on the x-axis.

- Create 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 (i.e. 5 line plots overlaid in different colors, containing your data from the files C4mm_800K.txt, etc.). Again, update the x-axis tick marks and labels indicate 1, 6, 12, 24, 36 and 48 hours.

All four of your plots should be appropriately annotated. Discuss these plots and explain their significance.

What to turn in:
Everything should be turned in on Canvas, in a single ".zip" or ".tgz" file containing all of the required items listed below.

Turn in all of the requested C++ functions and Jupyter notebooks as separate ".hpp", ".cpp", abd ".ipynb" files. Include a Makefile so that *all of your executables* can be built from within the same directory with only the "make" command, and so that "make clean" will remove all of the ".txt" data files, temporary ".o" object files, and ".exe" executables generated when running your project. *Do not include these ".txt", ".o" or ".exe" files in either this zip file or in your report.*

You will also discuss your project in a technical report. In this report you should:

- Use complete sentences and paragraphs.
- Explain the problems that were solved, and the mathematical approaches used on them.
- Describe your codes, including a discussion on any unique decisions that you had to make.
- Discuss all of your computed results. In this portion, you should include your plots, and you should paste any output (error, timing data) printed to the screen into the report.
- Answer all questions posed in this project.
- In your own words, explain why you found the results that you did, justifying them mathematically if possible.
- Include your Makefile, C++ and Python code as attachments *inside* the report file (or included inline), for grading.

This report should be in ".pdf" format.