ME 701 – Development of Computer Applications In Mechanical Engineering
Homework 8 –Due 11/17/2017

Instructions: All source code should be placed in a single TAR file of the form `lastname_firstname_hw8.tar`. All code should be compiled with a Makefile, as specified in Problem 3.

Problem 1 – Arrays and Matrices

Consider the (unimplemented) functions:

```
// C++
double interpolate(double x_new, double *x, double *y, int n, int order)
{
  /* ... */
}
! Fortran
double precision function interpolate(x_new, x, y, n, order)
    double precision, intent(in) :: x_new
    double precision, dimension(:), intent(in) :: x, y
    integer, intent(in) :: n, order
    ! ...
end function interpolate
```

The functions should return the value of `y_new` at `x_new` interpolated from the points `x` and `y`. The interpolation should be of the order given by `order` (first is linear, second is quadratic, etc.). The number of points used is `n`.
Your tasks:

1. Implement these functions in C++ and Fortran.
2. Produce corresponding Python modules using SWIG and f2py. Note, think carefully about the interpolate function and its signature.
3. Produce a Makefile that automates all of the compilation. Make sure that any system-dependent paths are *easily changed* by modifying a variable in your Makefile (e.g., for the Python path needed for SWIG).

Problem 2 – Monte Carlo

Monte Carlo integration really shines for multidimensional problems, which is why it works so well for integrating the 6-dimensional phase space of time-independent neutron transport problems. In this problem, we'll use a very simple test case to illustrate the power of Monte Carlo integration and assess the performance of C++ and Fortran using a variety of compiler options. For this problem, use the language you did not use for Problem 1.

Background: We've done numerical quadrature using pre-defined abscissa with corresponding weights. Monte Carlo is good at solving integrals by simulation. For example, consider a unit square with a concentric circle that touches the center of each of the square's for sides. If you threw

100 darts, the number of darts inside the circle divided by 100 would be pretty close to $\pi/4$ (think about that if it's not obvious!). We can use this approach to solve a variety of integrals. Consider any integrand of n variables $f(\mathbf{x})$ where $\mathbf{x} \in \mathbb{R}^n$. For simplicity, assume each component x_i of $\mathbf{x}$ is restricted to $x_i \in [0, 1]$, and also assume that $f \in [0, 1]$. Then we proceed as follows:

1. Generate random samples for each x_i, e.g., $x_i =$ `rand`(), where `rand` is the random number generator in your language of choice. Call the resulting vector of samples $\mathbf{x}'$.
2. Generate a random "rejection" sample $a =$ `rand`.
3. If $a < f(\mathbf{x}')$, add 1 to the "in the circle" bin (i.e., it's under the curve)
4. Otherwise, do another sample until we've done as many as we want.

Consider the integral

$$I_n = \int_0^1 dx_1 \int_0^1 dx_2 \ldots \int_0^1 dx_n (x_1 + x_2 + \ldots + x_n)^2 \,. \tag{1}$$

We want to integrate this function using

1. the midpoint rule (i.e., use the centers of multidimensional boxes)
2. Monte Carlo

It seems weird that the use of random numbers could ever be beneficial for this, but it turns out to be wickedly good for high-dimensional problems.

Deliverables:

1. Write a program in C++ **or** Fortran to compute the integral and the *relative, absolute error* (i.e., $|I^{appx} - I^{ref}|/|I^{ref}|$ with both quadratures for arbitrary n and an m, where m^n is the number of of evaluations of the integrand. (For the midpoint rule, m defines the number of divisions along a given axis.) I have provided template files for you to start with in both languages. The main program should accept n and m as command line arguments. It should output a file named `output.txt` that contains one line of the following form: `I I_mp I_mc`, where I, I_mp, and I_mc are the reference, midpoint, and Monte Carlo values.
2. Plot the *relative, absolute error* for $n = 1$, 3, and 10. Use enough integrand evaluations to obtain an absolute relative error of 0.001 or less, and use a log-log scale. You should use Python for running your program, reading the output file, and plotting the results. Name that file `hw4p2.py`.
3. For the largest n and m values you needed, time your Monte Carlo routine for the case of (1) no optimization and (2) "-O3" optimization and report the result in your summary PDF file. This could take some time (for the unoptimized case), so plan accordingly.