ME 701 – Development of Computer Applications In Mechanical Engineering
Homework 3 – Object-Oriented Programming, C++, and Modern Fortran – Due 10/20/2014
(Note the extended deadline)

Problem 1 – Object-Oriented Python

In class, we explored a bit of objected-oriented design for the problem of orthogonal expansions. Although this is a simple problem mathematically, it also has an intrinsic structure suitable for organization in object-oriented fashion.

Consider a function $f(x)$ defined over $x \in [a,b]$. The goal is to approximate $f(x)$ by the L-th order expansion

$$f(x) \approx= \sum_{l=0}^{L} \tilde{f}_l P_l(x), \tag{1}$$

where

$$\tilde{f}_l = \int_a^b w(x) P_l(x) f(x) dx\,, \tag{2}$$

and the functions $P_l(x)$ satisfy

$$\int_a^b w(x) P_l(x) P_m(x) dx = \delta_{lm} \tag{3}$$

for a weighting function $w(x)$. The function δ_{lm} is the Kronecker-δ, i.e., it equals 1 for $l = m$ and 0 otherwise. To evaluate the integral, we'll need the numerical quadrature covered in the last homework.

Alternatively, we can approximate discrete functions, i.e.,

$$f_i = \sum_{l=0}^{L} \tilde{f}_l P_{l,i} \quad i = 0, 1, \ldots, I\,, \tag{4}$$

where

$$\tilde{f}_l = \sum_{i=0}^{I} w_i P_{l,i} f_i\,. \tag{5}$$

In class, we found that a generic `OrthogonalBasis` class ought to have at least two methods

1. $\tilde{f} =$ `transform` (f)
2. $f_{\text{approx}} =$ `inverse_transform` $(\tilde{f})$

Then, a suggested class hierarchy was given, e.g.,

- `OrthogonalBasis` → `ContinuousBasis` → `LegendreBasis`
- `OrthogonalBasis` → `DiscreteBasis` → `DiscreteFourierBasis`

Deliverables:

1. Implement the abstract classes `OrthogonalBasis`, `ContinuousBasis`, and `DiscreteBasis`. Remember, abstract means they can't be used alone; rather, we need a concrete implementation.
2. Implement *at least* three, concrete subclasses (e.g., Legendre polynomials, Fourier transform, and discrete Fourier transform).

3. Test your bases using the following functions defined over $x \in [0, 10]$:
   (a) $x^2 + x + 1$
   (b) $\frac{1}{1+x^2}$
   (c) $J_0(x)$ (i.e., Bessel function of the first kind)

   Provide the L_2-norm of the error for each basis. For the discrete cases, use an `np.linspace` with 20 points. For all cases, go through at least a 10th-order approximation. I'd suggest plotting the error as a function of order.
4. Your very well-documented, very clean code.

BONUS # 1: Many times, we'll have a continuous function evaluated at discrete points (e.g., quadrature points) so that all of the bases can share some shared structure. An alternative you can explore is the following: pass into your basis a continuous function $f(x)$, then get $\tilde{f}$, and finally get a new, continuous function $f_{\approx}(x)$. You may find that "λ-functions" in Python are handy for this.

BONUS # 2: I will provide (soon, I hope) 10 or so random, discrete functions that are all "close" in some way (e.g., a 1-D representation of pictures of the same parking lot over the period of a day) . Your job is to define an orthogonal basis that provides the best approximation to those functions with as low an order as possible. To be explicit, find the best 4th order expansion, i.e., only 5 functions in the basis. This is a hard problem that will make you smart if done right. :).

HINTS: You might find my C++ code Detran to be of use; specifically, I've done something very similar to this exercise for my own research, and you can use the structure to guide what you do. Note, however, that it does not handle things like the bonus questions:
`https://github.com/robertsj/libdetran/tree/master/src/orthog`.

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.

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

$$I_n = \int_0^1 dx_1 \int_0^1 dx_2 \ldots \int_0^1 dx_n \sqrt{x_1 \times x_2 \times \cdots \times x_n}\,. \tag{6}$$

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 both C++ and Fortran to compute the integran and the *relative, absolute error* with both quadratures for arbitrary n and an arbitrary number of of evaluations of the integrand.
2. Plot the *relative, absolute error* for $n = 1$, 3, and 10. Use enough integrand evaluations to obtain an absolute relative error of 0.0001 or less, and use a log-log scale.
3. For the largest case, time your C++ and Fortran codes for the Monte Carlo for the case of (1) no optimization and (2) "-O3" optimization.

Problem 3 – The Sieve of Eratosthenes Revisited

Deliverables:

1. **In either C++ or Fortran**, implement a routine for providing all of the prime numbers less than n (where n is arbitrary). Your function should handle all values of n up to 10^5 in a reasonable amount of time (i.e., a few minutes at most).
2. I recommend one of the following structures:
   - `subroutine find_primes(n, p, np)`
   - `void find_primes(int n, int *p, int np)`

   where for Fortran, `p` is an allocatable array, and for both, `np` is the size of the array of primes. You can certainly try another approach if you don't like this one.
3. For $n = 10^5$, compare your time using C++ or Fortran to what you found with Python.