AtomicScaleSimulations_HW4

Created by Yu, Xiongjie, last modified by Yang, Paul on Oct 09, 2016

Questions, clarifications, comments, etc. about this homework can be posted on the course forum on Piazza.

In this homework, we will learn how to write a pseudo-random number generator (RNG) and how to assess the quality of a stream of random numbers from a particular RNG. We also study the technique of importance sampling. Random numbers and importance sampling are essential ingredients of Monte Carlo simulation techniques; in the next homework, you will modify your MD code to do Monte Carlo simulations.

Write Your Own Random Number Generator (RNG)

In this problem we will write a random number generator that generates a stream of uniformly distributed random numbers and one where the random numbers follow a Gaussian distribution.

Uniform Random Number Generator

In this section, you are going to write a RNG that uses the linear congruent generator (LCG) algorithm. An LCG generates a random number X_{n+1} by taking the current random number X_n and performing the following arithmetic:

$$X_{n+1} = (aX_n + c) \mod m$$

where m is the upper limit of possible values of X_n, a<m, and c<m. mod denotes the modulus. This algorithm requires choosing a number X_0 to start the stream, called the *seed*.

The quality of the LCG algorithm is very sensitive to the choice of these parameters. To begin with, your period can never be longer than m.

Write a function

```
def LCG(m,a,c,xn):
    #returns the next random number
```

that uses the LCG algorithm to produce the next pseudo-random number given the argument X_n.

Print out and submit the values of your generator for LCG(16,3,1,2) for 20 steps. How long does it take before it starts repeating? Are all the numbers between 0 and 15 represented?

For the rest of this assignment, use the parameters LCG(2^32,69069,1,0) in your LCG. Submit the first ten values from LCG with these parameters. Remember that in python exponentiation is denoted by 2**32.

Your LCG function should generate random integers between 0 and m. Often, we will want random numbers uniformly distributed between 0 and 1.
Process the output of your RNG to produce a stream of numbers in the range [0,1); print out the first ten values from your RNG.
It is also useful to have a stream of random numbers with mean 0 (i.e. the random numbers can take negative values).
Process the output of your RNG to produce a stream of numbers in the range [-0.5, 0.5). Print out the first ten values from your RNG.

Gaussian Random Number Generator

If you have a stream of uniformly distributed random numbers, as you just created above, it is possible to apply a transformation to get a stream of random numbers that obeys some other distribution.

In simulations of physical systems at thermodynamic equilibrium, it is useful to have random numbers drawn from a Gaussian distribution. There are a couple standard methods to accomplish this, and you should use textbooks and the internet to learn about and select an algorithm.

Implement an algorithm to generate random numbers drawn from a Gaussian distribution with mean and standard deviation as parameters that can be specified when the RNG function is called.
To show that your RNG produces the correct distribution, generate a stream of 10,000 numbers with mean 0 and standard deviation 1 and histogram the values. If normalized correctly, your histogram should match the function

$$\frac{1}{\sqrt{2\pi}}\exp\left(-\frac{x^2}{2}\right)$$

Produce a plot of your histogram compared with the Gaussian function.

The correct normalization of the histogram may not be obvious to you. If you have N random numbers, and your histogram contains N_B bins over a range [-L, L], you should multiply by the normalization factor N_B/(2LN).

Testing Random Number Generators

There are a lot of empirical methods to test random numbers. Here we will use one of the simple ones.

We start by thinking about the following scenario. If we have a RNG that claims to produce uniformly distributed random numbers on the interval [0,1), we can test the uniformity of the distribution by creating a histogram of the stream of random numbers. We create a series of evenly-spaced bins between [0,1) and we generate a stream of random numbers, binning each value appropriately. If the RNG stream is indeed uniformly distributed, we should observe the same number of random numbers in each bin.

Of course, if we actually try to do this empirically, we will find that not every bin gets the same number of counts. Instead, there will be fluctuations that cause some bins to get more counts than others. On the other hand, a RNG stream that is not truly uniform will also lead to a histogram with unequal counts in each bin. We want to distinguish between non-uniformity that occurs because of statistical fluctuations (i.e. having only a finite sample size) and non-uniformity due to a broken random number generator. We use the following metric to distinguish between these two cases:

$$\chi^2 = \sum_i (N_i - n_i)^2 / n_i$$

where N_i is the number of counts in bin i and n_i is the expected number of counts in bin i (the total number of points divided by the number of bins N_B for our case of the uniform distribution).

If n_i > 5 is true for all i, the χ^2 statistic approaches the eponymous chi-squared distribution with (N_B-1) degrees of freedom.

The statement of the goodness-of-fit can be simplified to this: If this χ^2 value is **greater** than some critical value, then with some quantifiable confidence this indicates that the generator of this stream is **not** what it is claimed to be (i.e. a uniform distribution of x in [0,1)).

We can try solving for these critical values numerically by integrating the probability distribution function for the chi-squared distribution, or for degrees of freedom up to 100 just use the tabulated values provided by NIST:

http://www.itl.nist.gov/div898/handbook/eda/section3/eda3674.htm

For this homework, let us choose the 90% confidence level for the hypothesis that the generator is not broken (random). Critical values of χ^2 should be taken from the upper-tail critical values column that says 0.90 (instead of 0.1).

One can also bin numbers in more than 1 dimension. For example, for 2 dimensions one could take pairs of numbers and bin them onto a grid. This is useful because it will expose correlations between successive numbers in the series. This is also a serious problem, because we generally use a RNG stream with the assumption that successive numbers are independent of one another.

Here, we will write functions that compute the χ^2 metric for histograms in two and three dimensions. Note that the chi-squared test becomes increasingly stringent in higher dimensions, and RNGs judged suitable for research purposes are generally required to pass the chi-squared test in 10 dimensions.

You may find yourself working with degrees of freedom greater than 100 (off the NIST table mentioned above), in which case you will have to calculate the values yourself by numerical integration or use the critical value calculator found at (I compared the first few values. It works.)

http://www.swogstat.org/stat/public/chisq_calculator.htm

For 90% confidence level, you should set the "probability" parameter to 0.9.

Write the functions

```
def CheckRandomNumbers1D(numberList,pointsPerDim):
    #takes a list of random numbers
    #generate a 1-dimensional histogram
    # and returns the chi_squared
    hist = numpy.zeros((pointsPerDim,)) + 0.0
    return chi_squared
```

```
def CheckRandomNumbers2D(numberList,pointsPerDim):
    #takes a list of random numbers
    #generate a 2-dimensional histogram
    # and returns the chi_squared
    hist = numpy.zeros((pointsPerDim,pointsPerDim)) + 0.0
    return chi_squared
```

and

```
def CheckRandomNumbers3D(numberList,pointsPerDim):
    #takes a list of random numbers
    #generate a 3-dimensional histogram
    # and returns the chi_squared
    hist = numpy.zeros((pointsPerDim,pointsPerDim,pointsPerDim)) + 0.0
    return chi_squared
```

that take a list of random numbers and the points per dimension and return the chi-squared value.

We have supplied you with 3 files of random numbers which you should download:
Generator1.dat, Generator2.dat, Generator3.dat
You may load them by calling numpy.loadtxt('Generator1.dat'), which returns a NumPy array containing the data.

Generate a stream of random numbers on the interval [0,1) using your LCG function and report the chi-squared values in 1,2, and 3 dimensions.
Perform the same tests on the random.random() module that comes with Python.
Perform the same tests on the streams from the 3 supplied files.
For each stream, state explicitly if the RNG passes the chi-squared tests.

Now, also write a function

```
def Plot2d(numberList,pointsPerDim):
    # makes a plot of the pairs of random numbers in 2d
    return None
```

that produces a plot of the numbers binned in 2 dimensions. You may find the command '''pylab.plot''' useful.

Run it on the first 1000 points of the three supplied datasets.

Use 1000 points for "Plot2d" only! For the chi-square test, use as many points as you need in order to satisfy n_i > 5.

Importance sampling

We wish to calculate the integral

$$I = \int_{-\infty}^{\infty} \mathrm{d}x \, \frac{\exp(-x^2/2)}{1+x^2}$$

which can not be done analytically.

First, you will compute the integral by numerical quadrature.

1. Choose finite limits of integration as [-L, +L], which you believe are a good approximation to the infinite limits.
2. Choose a number of grid points N (at least 10,000), which fixes your grid resolution Δx = 2L/N.
3. You will estimate the value of the continuous integral by performing a summation of the function at a discrete set of grid points:
   $$I = \int_{-\infty}^{\infty} \mathrm{d}x \, f(x) \approx \frac{1}{\Omega} \sum_{i=1}^{N} f(x_i) \; ,$$
   where Ω is the appropriate normalization involving N and L.
4. Find the correct expression for Ω, the normalization.
5. Write code to compute this integral with quadrature and report the result with your choice of parameters.

Secondly, we wish to compute this integral using Monte Carlo with importance sampling. The idea of importance sampling is that if we want to evaluate

$$I = \int_{-\infty}^{\infty} \mathrm{d}x \, f(x)$$

we can rewrite the integral as

$$I = \int_{-\infty}^{\infty} \mathrm{d}x \, p(x) \frac{f(x)}{p(x)} = \int_{-\infty}^{\infty} \mathrm{d}x \, p(x) g(x)$$

where p(x) is a probability distribution (i.e positive for all x and normalized such that

$$\int_{-\infty}^{\infty} \mathrm{d}x \, p(x) = 1$$

and

$$g(x) = \frac{f(x)}{p(x)}$$

is the estimator.

For our integrand

$$f(x) = \frac{\exp\left(-\frac{x^2}{2}\right)}{1+x^2}$$

we will choose

$$p(x) = K \exp\left(-\frac{x^2}{2\alpha}\right) \; ,$$

where K is chosen so that p(x) is a valid probability distribution (i.e. normalized correctly).

Determine an expression for K that normalizes p(x).

The utility of doing this is that we know how to sample a Gaussian distribution using a Gaussian random number generator like the one you wrote above. Thus, for Monte Carlo integration with importance sampling, we will sample random configurations distributed according to the function p(x), and we will evaluate the integral I by computing the expectation value of the estimator g(x):

$$I = \langle g(x) \rangle_{p(x)} \; .$$

Using the functional forms of p(x) and f(x) given above, write down an analytic expression for the estimator g(x).

Also write down an expression for the estimator of the variance of I computed with Monte Carlo integration sampling the distribution p(x).

Compute the value of the integral with Monte Carlo integration using your estimator for the mean and your Gaussian distribution of random numbers. Also report the variance of your answer using the appropriate variance estimator.

```
def CalcIntegral(alpha):
    #calculates entire integral
    #returns the integral and the variance!
    return (mean,variance)
```

that returns both the mean AND variance. You may have to use python code that you wrote in HW1 to compute the variance. Notice, that for different alpha's you are using different importance functions and hence will get a different variance. Graph the variance versus alpha and submit this with your code.

You should optimize your importance sampling by finding the parameter α that minimizes the variance. Choose at least 8 different values of α between 0 and 2 and report your estimate of the integral and the variance for each.
Submit a plot of the variance as a function of α.
Based on this plot, report your optimal choice of α.

If the variance were infinite for certain values of α, we would not know it from the results above.

As a first test, run your simulation again with 4 times as many steps. If your variance is well defined, then we know that the variance should be effectively independent of how long you run it. If this turns out not to be the case, then you should worry that you have infinite variance! Submit a graph that graphs alpha versus (the ratio your calculated variance with 4× as many steps to the original calculated variance)). Also indicate where you have infinite variance.

Calculate the variance analytically, expressing your answer as a function of α.
State the range of values of alpha for which the variance is infinite.

Now that you have located values of α where the variance is infinite, we will look to see how this manifests itself in the numerical data. Make a trace plot of g(x,α) versus sample number for a value of α that has infinite variance and a value of α that has finite variance. Describe the qualitative difference between these two situations.

No labels