PHYC 2050 Class Notes

Thomas J. Duck

Department of Physics and Atmospheric Science

Dalhousie University

tom.duck@dal.ca

March 24, 2010

CHAPTER

SIX

MONTE CARLO METHODS

We cannot always launch the ball at the same speed, nor can we always achieve the same launch angle. The wind might also change from time to time. What range of drive distances might we expect given this variability? The answer is far from trivial because the parameters we are concerned with interact non-linearly.

The "Monte Carlo" method is to vary all of the parameters randomly (within reasonable limits) in a collection of simulations, and then determine the statistics of the output results. The overall approach invokes the image of a gambler's roll of the dice. On average the gambler will lose, although any one throw might be lucky. Similarly, although any one swing of the golf club might produce great distance, there will be an average distance we can expect to attain, and a range of distances over which the golf balls would most often land.

6.1 Probability Distributions

A probability density function (pdf) that describes the random variability in many kinds of physical system is the "Gauss" normal distribution or "bell curve". The Gauss distribution is defined by

$$P(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp[-(x-\mu)^2/(2\sigma^2)]$$

where μ is the mean value σ is the standard deviation.

The `matplotlib.mlab.normpdf()` function allows us to quickly calculate and then plot a few sample Gauss distributions. Let's try plotting up two distributions with the same mean of 5, but standard deviations of 2 and 0.5:

```
#! /usr/bin/env python

# gauss.py

import numpy
import pylab, matplotlib

x = numpy.arange(0,10,0.01)
y1 = matplotlib.mlab.normpdf(x,5,2)
y2 = matplotlib.mlab.normpdf(x,5,0.5)

pylab.figure(figsize=(6,4))

pylab.plot(x,y1,label='mean=5, sd=2')
pylab.plot(x,y2,label='mean=5, sd=0.5')

pylab.xlabel('x')
pylab.ylabel('P(x)')

font = matplotlib.font_manager.FontProperties(size='small')
legend = pylab.legend(prop=font)
legend.draw_frame(False)

pylab.show()
```

The output of the program is given in Figure 6.1. The figure shows that the highest probability for both curves is the mean value of 5, as expected. The distribution with the larger standard deviation is wider; i.e., has a greater probability for events further away from the mean value.

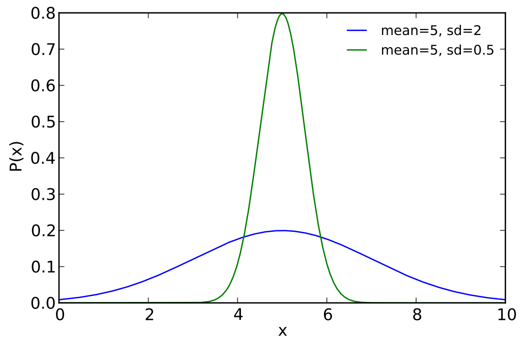

Figure 6.1: Two different Gauss normal distributions. [gauss.py]

An important related concept is the accumulated probability. If $P(x)$ is the probability density function, then the accumulated probability

$$A(x) = \int_{-\infty}^{x} P(x')dx'$$

gives the total probability between $-\infty$ and x. Accumulated probabilities for the Gauss distributions in Figure 6.1 are given in Figure 6.2.

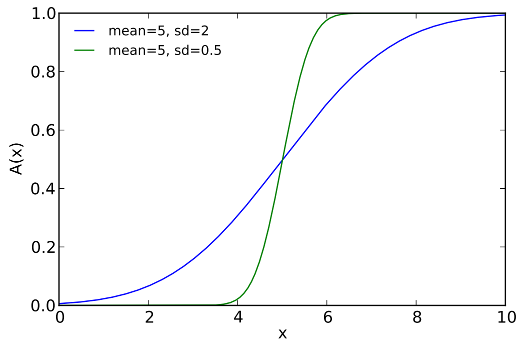

Figure 6.2: Accumulated probability distributions for the bell curves given in Figure 6.1. [gauss_accumulated.py]

In both cases, the accumulated probability for $x < 0$ is nearly zero, and for $x < 10$ it is nearly 1, as we may have expected. What happens in between is sensitive to the shape of the underlying probability density function. The accumulated probability doesn't convey any extra information (it's derivative is just the Gauss distribution), but it is a useful concept, as we shall see.

6.2 Pseudo-Random Number Generation

There are a variety of algorithms that can be used to generate pseudo-random numbers. One such algorithm is the "Linear Congruential Generator", which generates a series of random integers defined by the recursive equation

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

where m is the modulus ($m > 0$), a is the "multiplier" ($0 < a < m$), and :math'c' is the "increment" ($0 < c < m$). Given the seed value X_0, the equation generates a series of pseudo-random numbers between 0 and m.

How does it work? Consider a walk around a circle of circumference m, as shown in Figure 6.3. We begin by taking c, steps in the clockwise direction, followed by X_n steps a times. If we travel around many times, then where we end up on the circle is essentially random. The value X_{n+1} is taken to be the number of steps measured clockwise from the starting point to the final position (without laps).

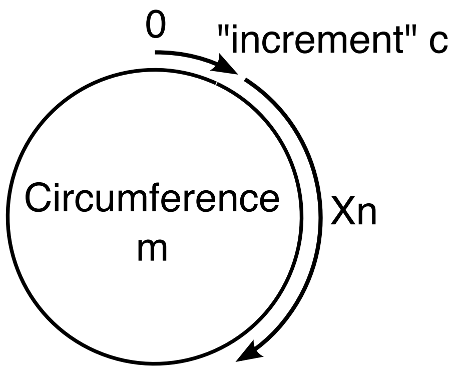

Figure 6.3: Schematic representation of the Linear Congruential Generator. Starting at position 0, take $c + aX_n$ steps around the circle. X_{n+1} is the final position measured clockwise from 0.

The numbers are only pseudo-random because the sequence from the Generator repeats, as it must if we are constrained by finite memory. Here is an example implementation for $m = 16$:

```
>>> m,a,c = 16,3,5
>>> def rand(seed=None):
...     if seed!=None:
...         rand.x = seed
...     rand.x = (a*rand.x+c) % m
...     return rand.x
...
```

Starting with a seed value of 0, this produces the series $5, 4, 1, 8, 13, 12, 9, 0, \ldots$ which repeats since the seed value is seen again.

A good choice of parameters m, a, and c, are needed if a quality series of random numbers is to be generated. By "quality", I mean a series of numbers that satisfy statistical tests for randomness, such as a uniform probability density function and no correlations. The glibc library uses $m = 2^{32}$, $a = 1103515245$, and $c = 12345$, and returns only bits

0 through 30. The results it produces are OK, but more sophisticated generators can do better. Nevertheless, they all have the same property of requiring a seed to start.

The program below creates a pseudo-number series between 0 and 1 with our Linear Congruential Generator, and plots it together with a histogram representing the probability density function. Figure 6.4 displays the results. The data are uniformly distributed in the left panel, and this is confirmed by the flat probability density found in the right panel. This kind of randomness, where all values have equal probability of occurring, is called “white noise”.

```python
#! /usr/bin/env python

# pseudorandom.py

import numpy
import pylab

m,a,c = 2**32, 1103515245, 12345  # glibc values

def rand(seed=None):
    """Returns random number between 0 and 1
    using the linear congruent generator."""
    if seed!=None:
        rand.x = seed
    rand.x = (a*rand.x+c) % m
    return float(rand.x)/m

rand(0)  # Seed

# Generate the random numbers
x = numpy.array([rand() for i in range(1000000)])

# Plot
pylab.figure(figsize=(10,4))

pylab.subplot(121)
pylab.plot(numpy.linspace(0.,1,len(x))[::200],x[::200],'+')
pylab.xlabel('Index (millions)')
pylab.ylabel('x')

pylab.subplot(122)
pylab.hist(x, 30, normed=True)
pylab.xlabel('x')
pylab.ylabel('P(x)')

pylab.show()
```

How can we generate pseudo-random numbers with a specific distribution? Figure 6.5 illustrates the idea. We first get to choose the probability density function, and from it calculate the accumulated probability. Next, white noise values between 0 and 1 are generated and taken to be the accumulated probabilities. Each white noise value maps to an x-value specified by the accumulated probability curve. The series of x-values that result are distributed according to the underlying probability density function.

Fortunately, all this work is already done for us. The `numpy.random` module has functions that can generate random numbers for different probability density functions. For example, here is how we could generate Gauss-distributed noise:

```python
>>> import numpy, numpy.random
>>> numpy.random.seed()
>>> x = numpy.random.normal(5,0.5,1000000)
```

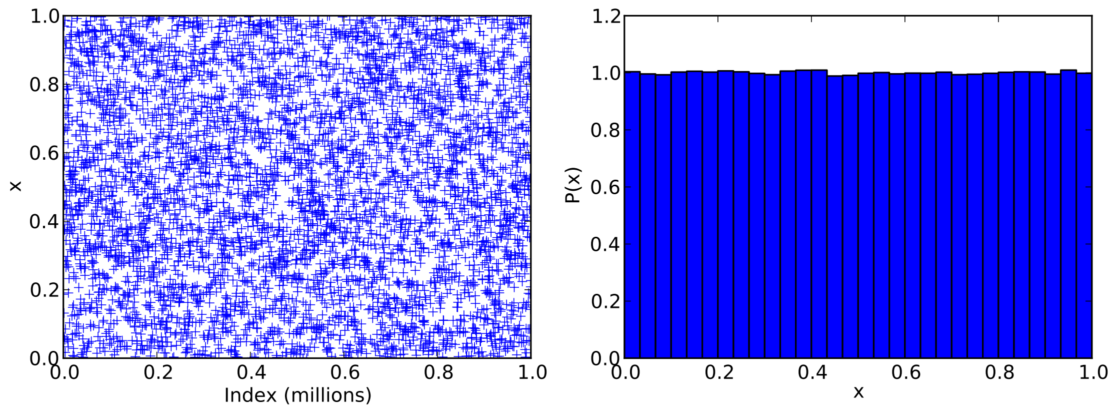

Figure 6.4: Results from one million iterations of the Linear Congruential Generator. The left panel gives every 200th value (normalized between 0 and 1), and the right panel gives the probability density function in histogram form. Both plots confirm the generator produces "white noise". [pseudorandom.py]

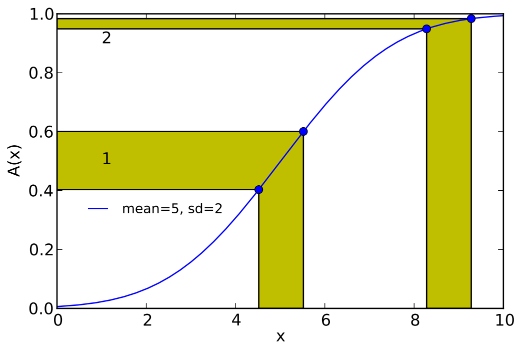

Figure 6.5: The generation of Gauss-distributed noise from a white noise signal. Random accumulated probabilities are generated first, and are mapped to corresponding x-values via the accumulated Gaussian probability curve. For the given curve there is good chance of randomly selecting an accumulated probability between 0.4 and 0.6 (Region 1), and therefore a high probability for x-values between 4.5 and 5.5. There is a much lower probability for randomly selecting an accumulated probability between 0.95 and 0.98 (Region 2), and so low probability for x-values between 8.3 and 9.3. Thus, the x-values are distributed according to the underlying Gauss probability density function. [gauss_accumulated2.py]

```
>>> print x
[ 4.34527533  5.05385408  4.12150776 ...,  5.76390538  5.69932093
  5.5023789 ]
```

After the imports, we set the initial value for the random number generator using `numpy.random.seed()`. If no arguments are given, then a number based upon the current time is used. If, for some reason, you wanted to generate the same set of random numbers multiple times, you could provide a numerical argument to the seed function as the initial value.

The `numpy.random.normal()` function is called next. It requires the mean value and standard deviation followed by the number of values we want to generate (defaults to 1). In this case, an array of 1000000 numbers is returned. We can quickly verify the mean value and standard deviation are what we expect using some of the statistical functions in `numpy`:

```
>>> print numpy.mean(x), numpy.std(x)
5.00035013494 0.500745264241
```

Close enough. Figure 6.6 gives the data series and corresponding histogram. The random data are confined near the mean value of 5, and are mostly spread between 4 and 6 (i.e., two standard deviations from the mean). As we might have expected, the histogram looks a lot like the more narrow of the two distributions in Figure 6.1.

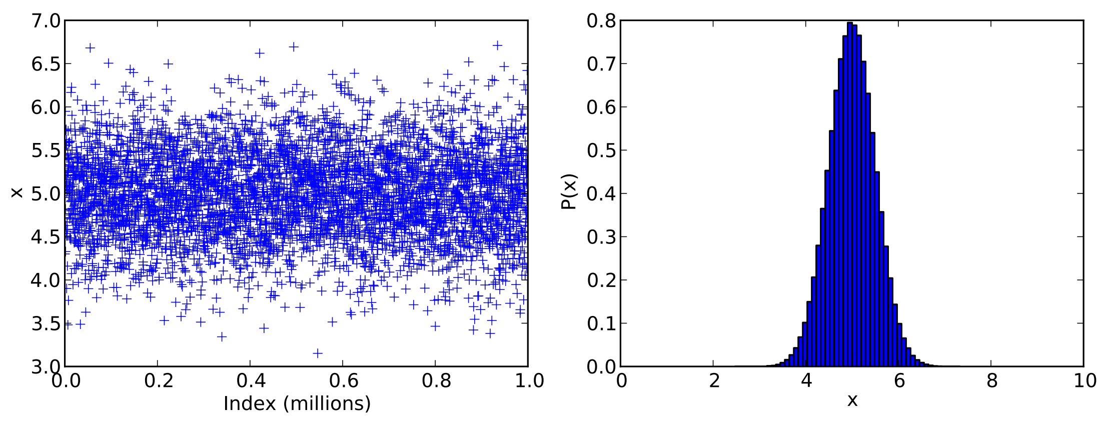

Figure 6.6: Results from the generation of one million Gauss-distributed random numbers. The left panel gives every 200th value, and the right panel gives the probability density function in histogram form. The mean value and standard deviation we specified were 5 and 0.5, respectively. [gauss_random.py]

There are other distributions functions observed in nature beyond Gauss. For example, the gamma distribution is defined by

$$P(x) = x^{k-1}\frac{\exp(-x/\theta)}{\theta^k\Gamma(k)}$$

where k is the "shape parameter", θ is the "scale parameter", and Γ is the so-called "gamma function" which is just a continuous version of the factorial function. Gamma-distributed noise for a specific choice of parameters is shown in Figure 6.7. The gamma distribution produces values greater than 0 with a long-tailed probability density function, quite unlike either white noise or Gauss.

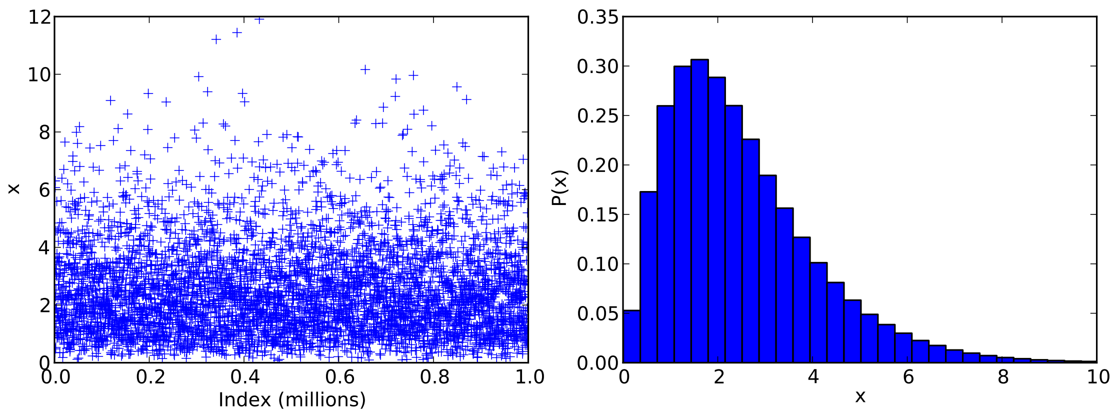

Figure 6.7: Results from the generation of one million Gamma-distributed random numbers with $k = 2.5$ and $\theta = 1$. The left panel gives every 200th value, and the right panel gives the probability density function in histogram form. [gamma_random.py]

6.3 Golf Monte Carlo

Now that we can generate random numbers following specific distributions, let's use them to see how physical variability affects the drive distance in golf. Consider the following program:

```
#! /usr/bin/env python

# golf_monte_carlo.py

import numpy
import numpy.random; numpy.random.seed()

import pylab
import matplotlib.mlab

import cgolf


cgolf.h = 0.01


# Define the mean and standard deviation for each parameter
Vmean, Vsd = 50., 2.
theta_mean, theta_sd = 18., 2.
rpm_mean,rpm_sd = 3000., 150.
Umean, Usd = 3., 1.

# Define a function that returns the wind speed
n = 0   # Countdown to next wind change
U = 0   # Current wind speed (m/s)
def get_U(x,z):

   global n, U

   if n == 0:
        # Get the new wind value
        U = numpy.random.normal(Umean,Usd)

        # Determine how many steps before the next wind change.
        # Let the change time be governed by the gamma distribution.
        n = int(numpy.ceil(numpy.random.gamma(2.5,1)/cgolf.h))

    else:
        n -= 1 # Count down

    return U


# Store the results of each simulation in a list
distances = []


# Run the simulation subject to randomness in each of the parameters
for i in range(10000):

    V = numpy.random.normal(Vmean,Vsd)
    theta = numpy.random.normal(theta_mean, theta_sd)
    rpm = numpy.random.normal(rpm_mean, rpm_sd)

    distances.append( cgolf.get_carry_distance(
          *cgolf.get_trajectory(V,theta,rpm,get_U=get_U)) )


# Calculate the drive distance
distance_mean = numpy.mean(distances)
distance_sd = numpy.std(distances)


# Plotting

pylab.figure(figsize=(6,4))

# Plot a histogram (it helpfully returns the distribution too)
distribution, x, patches = pylab.hist(distances,51,normed=1)

# Overlay a Gaussian 'normal' probability distribution function
x2 = numpy.arange(x[0],x[-1],0.1)
y2 = matplotlib.mlab.normpdf( x2, distance_mean, distance_sd )
pylab.plot(x2, y2, 'r--', linewidth=2)

# Labels and text
pylab.xlabel('Distance (m)')
pylab.ylabel('Probability')
pylab.text(120, 0.035, '%.0f +/- %.1f m' % (distance_mean, distance_sd),
           color='r')

pylab.show()
```

After the imports, we define the mean and standard deviations for the launch speed, angle, spin rate, and wind speed We assume that each of these parameters is governed by a Gaussian distribution, and have assigned what are hopefully reasonable values for each of the parameters. The true variability, however, will vary from golfer to golfer.

Next we define a function that will provide the wind values. We choose to parameterize the time between changes in the wind speed as a gamma distribution. This is not based on any measurements, and a better model would be based on a measured distribution function. That being said, it will serve its purpose here acceptably.

The `get_U()` function introduces the `global` keyword. Declaring a variable from outside the function as global makes it so that changes to the variable are seen outside the function's *scope*. Without declaring them as global, any changes we made would be lost once the function exited.

Now on to the simulation. We create a distances list to store the results of 10000 simulations. For each simulation we get a new value for the launch speed, angle, and spin rate. The wind speed is provided by the `get_U()` function we defined earlier. The carry distance is determined from a simulated trajectory and stored in the list.

Finally, we create a histogram for the carry distances, which is shown in Figure 6.8. The figure reveals that the mean distance is 154 m. That is actually quite close to what we would have achieved using the mean value for each of the parameters. The simulation also reveals that the standard deviation is about 11 m. So, 66% (95%) of the golf balls will land between 143 and 165 m (132 and 176 m). That's quite a bit of variance, which tells you that the pros who drive the same distance every time are very consistent.

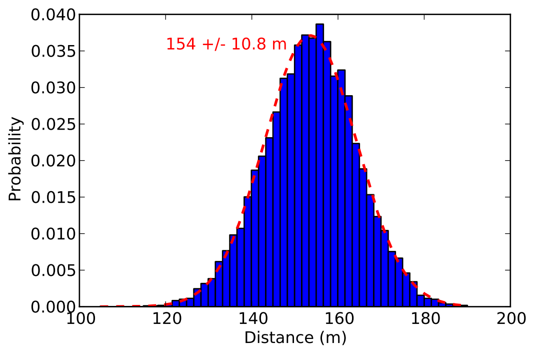

Figure 6.8: The probability density function for drive distances from the golf Monte Carlo simulation. [golf_mote_carlo.py]

The distribution of distances in Figure 6.8 is fairly close to Gaussian, as evidenced by the Gauss distribution plotted over top, although there are subtle differences. It didn't have to be this way, however. If you turn off all sources of variability other than in the launch angle (i.e., set their standard deviations to something small like 1×10^{-3}), you will obtain a highly skewed distribution of carry distances. Try it!

What science can we do with this model? Try changing some of the parameters to see where most of the variance in distances comes from. That will tell you where the most improvement from consistency can be gained (for a golfer with this as a starting point). You can also do some comparisons between gusty and steady days, and use the information to help plan your strategy on the golf course.