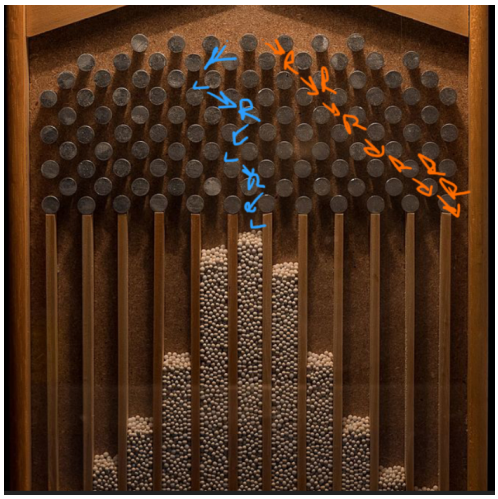
Chance
Mathematical Programming with Python

https://people.sc.fsu.edu/~jburkardt/classes/mpp_2023/chance/chance.pdf



Each ball "chooses" a random path, and yet there is a pattern in the results

Chance

- *Chance arises when we don't understand or control a situation;*
- *Chance helps us select cases at random.*
- *We may model chance with a probability model.*
- *A model predicts average and variance of chance values.*
- *For a random process with rewards, we can get expected value.*
- *A standard tool is the Monte Carlo method.*
- *Integrals over irregular or high dimensional regions can be approximated this way.*
- *Python functions can model many random processes.*
- *Combinatorial objects like subsets, permutations, combinations, can be randomly selected.*

1 Initializing the random number generator

There has been a recent change to the procedures by which a Python user requests random objects. In the past, for instance, a typical request for a single real random number would look something like:

```
r = np.random.random ( )
```

You may still see this sort of usage in recently written programs and textbooks.

However, the developers have created a new, flexible and powerful interface to the generation of random objects, which we will prefer in this course. Using this interface requires an additional step in which the random number generator is specified. We will always use the default generator, and so a typical program using random objects might begin as follows:

```
from numpy.random import default_rng  # Defines the random number generator
import numpy as np

rng = default_rng ( )  #  Gives the random number generator a name.
```

Aside from `random()`, the `numpy.random` library includes a generous selection of functions to return randomly selected normal real numbers, integers, permutations, subsets, and other objects, as we will see.

2 Using the seed

The call to `default_rng()` allows the system to initializes the random number generator. If, instead, we wish to control how the random number generator starts, we can supply a *seed*. If we run our code twice, using the same initial seed, then each time we will compute the exact same sequence of "random" values:

```
  seed = 123456789
  rng = default_rng ( seed )
```

If you specify the seed, you are saying that you want your program to reproduce the same random sequence every time you run it.

```
def use_seed ( ):
  from numpy.random import default_rng
  import numpy as np

  seed = 123456789
  rng = default_rng ( seed )
  for i in range ( 0, 5 ):
    print ( rng.random ( ) )
  return
```

After generating some random values, you might want to backup and generate the same sequence again:

```
def repeat_seed ( ):
  from numpy.random import default_rng
  import numpy as np

  seed = 987654321
  rng = default_rng ( seed )
  x1 = rng.random ( size = 3 )
  print ( 'x1 = ', x1 )
  x2 = rng.random ( size = 3 )
  print ( 'x2 = ', x2 )
  rng = default_rng ( seed )  #  Restart the generator
  x3 = rng.random ( size = 5 )
  print ( 'x3 = ', x3 )

  return
```

3 Uniform random real numbers

Python's most commonly used random number generation is `rng.random()`, which returns a single number or, if `size` is specified, a `numpy` array of the given size, filled with real numbers selected uniformly at random from the interval $[0, 1]$. "Uniformly" means every number in the interval is equally likely to be chosen.

```
  r = rng.random ( )  # returns a number, NOT an np.array
  s = rng.random ( size = 1 )  # an np.array of size 1
  v = rng.random ( size = n )  # an np.array of size n, a vector
  A = rng.random ( size = [ m, n ] )  # an np.array of size m x n, a matrix
```

Often, we want our random numbers to be in some other interval, $[a, b]$. This could be done by calling `rng.random()` and then adjusting the data

```
a = 10
b = 15
x = a + ( b - a ) * rng.random ( )  #  Returns a number
v = a + ( b - a ) * rng.random ( size = 5 ) # Returns an np.array
```

but a better solution is to call `rng.uniform()`:

```
a = 10
b = 15
x = rng.uniform ( low = a, high = b )  #  Returns a number
v = rng.uniform ( low = a, high = b, size = 5 ) # Returns an np.array
```

4 Normal random real values

You are probably familiar with the idea of the standard normal distribution, for which a histogram of many random samples produces a bell-shaped curve, which has a maximum at $x = 0$ (the "mean"), gradually dropping to near zero as x moves toward $\pm\infty$, with a standard deviation $\sigma = 1$.

To generate n samples from this distribution, call

```
v = np.random.standard_normal ( size = n )
```

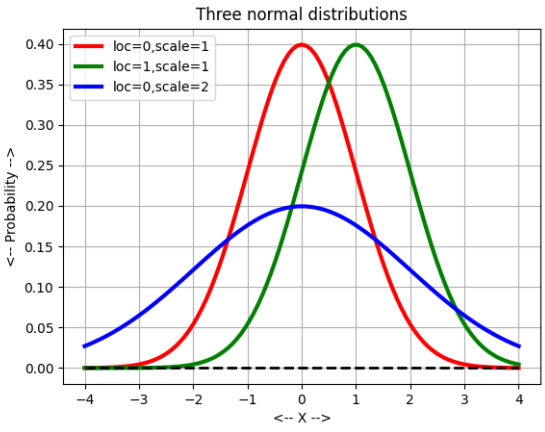
or, to generate samples with a mean value of `mu` and standard deviation `sigma`, call

```
v = np.random.normal ( loc = mu, scale = sigma, size = n )
```

which is equivalent to:

```
v = mu + scale * np.random.standard_normal ( size = n )
```

The `loc` parameter determines the average value of the random numbers, while the `scale` parameter controls the spread, that is, how far from the average a typical value might be. To get a feeling for how these parameters affect the result, consider the following plot of three normal distribution functions:



The `loc` *and* `scale` *parameters affect the placement and shape of the normal curve.*

5 Positive Definiteness Check With Random Vectors

Some algorithms in scientific computing are only guaranteed to work if a given matrix A is symmetric and positive definite, a characteristic that is sometimes abbreviated as **SPD**. An $n \times n$ real matrix A is SPD if,

and only if, it is true that $x'Ax > 0$ for all real, nonzero n vectors x. For a given matrix A, then, we can perform a crude check by computing this quantity for a large number of test vectors x.

If might seem that we could generate a good test vector x by calling `rng.random(size=n)`. However, all the entries of x will be positive. If all the entries of A are also positive, then $x'Ax$ will be positive, whether A is SPD or not. So our first choice for x vectors is not "random enough". To generate a more representative sample, we can use `rng.standard_normal(size=n)`, which will generate vectors with both positive and negative signs; also the magnitudes of some entries may be greater than 1.

Here is a sample code that checks the second difference matrix (which is SPD) and the Fibonacci matrix (which is not). If we had used the wrong random vectors, the Fibonacci matrix will be reported as probably SPD:

```
  m = 5
  n = 5
  A1 = dif2_matrix ( m, n )
  A2 = fibonacci1_matrix ( n, 1.0, 2.0 )

  for A, name in [ [ A1, 'dif2' ], [ A2, 'fibonacci' ] ]:

    fail = 0
    for test in range ( 0, 100 ):
#     x = rng.random ( size = n )           # Bad choice
      x = rng.standard_normal ( size = n )  # Better choice
      xtAx = np.dot ( x.T, np.matmul ( A, x ) )
      if ( xtAx <= 0.0 ):
        fail = fail + 1
    if ( 0 == fail ):
      print ( name + ' matrix passes SPD test' )
    else:
      print ( name + ' matrix fails SPD test' )
```

This example suggests that you have to be careful to ensure that your random quantities really do sample all the possible values you are considering!

6 Uniform random integers

If we are trying to pick a random day of the year, a random card, or a random number of children in a family, then we really need an integer result from our random number generator. The appropriate function to use is called `rng.integers()` and we can think of it as having the form:

```
  rng.integers ( low = a, high = b, size = ?, endpoint = True/False )
```

You can see that we need to specify an interval $[a, b]$ from which our integers will be selected. Being Python, there is a temptation to interpret this interval as $a \leq n < b$, in other words, not to include the upper endpoint as a possible result. This is the default behavior. However, if you include the argument `endpoint = True`, you are requesting that the interval include the final endpoint `b`.

The `size=?` argument is optional. If omitted, a single number is returned; otherwise, (even if `size=1`), you will get a numpy array of the given shape and size.

Here are four different ways to get 5 numbers out of an interval of 100 values:

```
v1 = rng.integers ( low = 0, high = 100, size = 5 )                   # 0 <= n < 100
v2 = rng.integers ( low = 0, high = 99, size = 5, endpoint = True )   # 0 <= n <= 99
v3 = rng.integers ( low = 1, high = 100, size = 5, endpoint = True ) # 1 <= n <= 100
v4 = rng.integers ( low = 1, high = 100, size = 5 )                   # 1 <= n < 101
```

Similarly, you could request a binary string of length 50:

```
b1 = rng.integers ( low = 0, high = 2, size = 50 )                    # 0 <= n < 2
b2 = rng.integers ( low = 0, high = 1, size = 50, endpoint = True )   # 0 <= n <= 1
```

Note that if `rng.integers()` returns a list, it is quite possible that some values will be repeated. This is because we are performing what is technically called "selection with replacement"; even if a value is picked for an entry of our list, it can be picked again for a later entry of the same list. This happens, obviously, in the case of the binary string, where you only have two choices for 50 slots; but it also can happen when we are picking 5 values from an interval of 100. This means that `rng.integers()` would **not** be a good choice if you are trying to model a card game, in which the 52 items can each be chosen only once or not at all.

7 Random Integers Solve Newton's Dice Problem

According to Wikipedia: *In 1693, Samuel Pepys and Isaac Newton corresponded over a problem posed to Pepys by a school teacher named John Smith. The problem was:*

Which of the following three propositions has the greatest chance of success?

1. *Six fair dice are tossed independently and at least one 6 appears.*
2. *Twelve fair dice are tossed independently and at least two 6s appear.*
3. *Eighteen fair dice are tossed independently and at least three 6s appear.*

Pepys thought that the third outcome had the highest probability, but Newton thought it was the first outcome

While the correct answer can be determined using probability, we can quickly come up with a computational approach that helps us to guess what the answer will be. We can repeat each experiment 1000 times, and report the frequency with which the desired number of 6's occur. Here's how we would do it for the first case:

```
freq = 0
test_num = 1000
for test in range ( 0, test_num ):
  dice = rng.integers ( low = 1, high = 6, size = 6, endpoint = True )
  sixes = np.where ( dice == 6 )
  if ( 1 <= np.size ( sixes ) ):
    freq = freq + 1
print ( '  Case 1 probability estimate is ', freq / test_num )
```

The exercises ask you to estimate the probabilities for all three cases, and decide who was right, Newton. Pepys, or neither of them!

8 Selection without replacement

If you want to guarantee that your random list does not have any repeated values, you are actually asking for a random *subset* of the integers from the range $a \leq k < b$ or $a \leq k \leq b$. For instance, in most card games, you draw several cards from a shuffled deck. You keep each card before drawing the next. This means it's impossible for you to draw the same card more than once; the corresponding random function needs to take this fact into account. In Python, we can select a few items from a set at random by using the `rng.choice()` function.

This function has the form

```
subset = rng.choice ( my_set, n )
```

where `my_set` is a list of values, `n` is the desired number of selections, and `subset` contains the chosen values. For this function, the value of `n` can't be larger than the number of elements in the set. Here is are two examples, for small and large sets:

```
small_set = range(10,15)
subset = rng.choice ( small_set, 4 )
big_set = range(0,1000)
subset = rng.choice ( big_set, 5 )
```

Remember that the `range()` function omits the last value!

9 Estimating the probability of a straight in poker

A standard poker deck consists of 52 cards. The cards are organized into four suits: hearts, clubs, diamonds, and spades. Each suit contains 13 cards, having the ranks 1 through 13. After the deck is randomly shuffled, a player received 5 cards from the deck. Certain arrangements of cards are more valuable than others, and the players proceed to bet on who has the best hand.

One fairly rare arrangement is known as a *straight*. It is any hand of 5 cards which can be arranged to form a sequence, such as 4, 5, 6, 7, 8. In this case, only the ranks of the cards matter, while the suits may be different.

We wish to estimate the probability of getting a straight by simulating a random poker hand. It is natural to start this process by defining `deck` as a list of the values 1 thought 52, and then using `rng.sample()` to select from `deck` 5 different values:

```
deck = range ( 1, 53 )
hand = rng.choice ( deck, 5 )
```

If our hand is the values {7, 8, 9, 10, 11} then we can see right away that we have a straight. However, there are 4 cards of rank 7, namely 7, 7+13=20, 7+26=33, and 7+39=46, representing the 7s of hearts, clubs, diamonds and spades. We need to be able to turn the card index into a rank. That's easy, because as you can guess, it's almost just the remainder after division by 13. However, the cards 13, 26, 39 and 52 will end up with rank 52 unless we do a little quick adjusting:

```
rank = ( ( hand - 1 ) % 13 ) + 1
```

You should convince yourself that this adjustment correctly ranks all 52 cards.

If this was a real poker game, the next thing you might do is sort your cards by rank. We can do the same thing in Python with the `.sort()` method:

```
rank.sort()
```

And now we can easily check whether we have a straight:

```
straight = True
for i in range ( 0, 5 ):
  if ( rank[i+1] != rank[i] + 1 ):
    straight = False
    break
```

So now we have the tools we need to estimate the probability of a straight. We simply deal out a large number of hands, count the number of occurrences of a straight, and take the ratio. The exact value is $10,240/2,598,960 \approx 0.0039$ which means that about 4 times in 1000 you would be dealt a straight.

The exercises ask you to verify this claim.

10 The PDF, CDF, and ICDF for a random process

If we know that some random process is described by a given CDF function, then we should be able to determine the PDF by differentiation. We may also be able to compute ICDF, the inverse CDF. For instance, for our fly simulation, we have, for $0 \leq x \leq 1$:

$$\begin{array}{ll} \text{pdf(x):} & p = 2x \\ \text{cdf(x):} & c = x^2 \\ \text{icdf(c):} & x = \sqrt{c} \end{array}$$

To summarize,

- cdf(x) tells us the probability that the variable is between the lower limit and x.
- pdf(x) tells us the relative probability of the variable having the value x.
- icdf(c) tells us the variable value x whose CDF is equal to c.

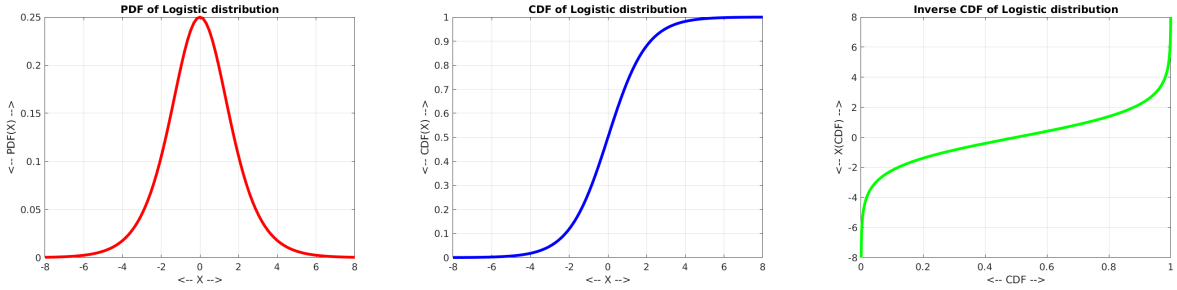
As another example, consider the *logistic distribution*, defined for $\infty < x < +\infty$ as:

$$\text{pdf(x):} \quad p = \frac{e^{\frac{a-x}{b}}}{b \cdot (1+e^{\frac{a-x}{b}})^2}$$

$$\text{cdf(x):} \quad c = \frac{1}{1+e^{(\frac{a-x}{b})}}$$

$$\text{icdf(c):} \quad x = a - b \log(\frac{1-c}{c})$$

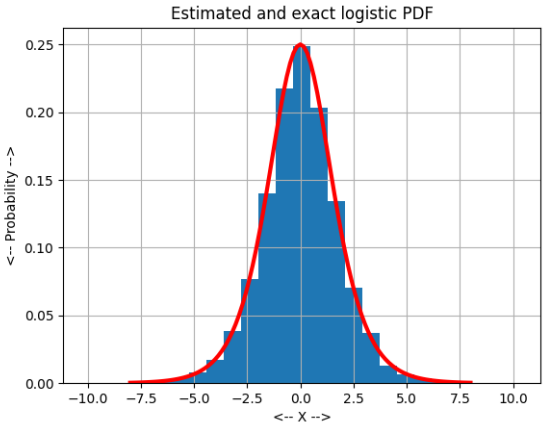
Here are the plots of these three functions over the partial domain $-8 \leq x \leq +8$, assuming parameters $a = 0, b = 1$;



PDF, CDF, ICDF for logistic distribution.

If we have a formula for the inverse CDF, then we create sample values from the corresponding distribution. For instance, we can generate 10,000 samples from the logistic distribution and histogram the result:

```
from numpy.random import default_rng
rng = default_rng ( )
a = 0.0
b = 1.0
c = rng.random ( size = 10000 )
x = logistic_icdf ( c, a, b )
plt.hist ( x, bins = 20, density = True )
x2 = np.linspace ( -8.0, 8.0, 101 )
pdf = logistic_pdf ( x2, a, b )
plt.plot ( x2, pdf, 'r-' )
```



Our random samples follow the logistic distribution.

11 Where's that fly (probably)?

In the chapter on Simulation, we discuss a problem in which a fly lands on a circular plate whose diameter is 1. In that discussion, we explored how to use simulation to estimate the average distance from the fly to the center of the plate. We used the following algorithm to randomly place the fly:

```
Purpose:
  Place a fly at random in a circle of radius 1
Compute
  r1, r2 = two uniform random values
  (r,theta) = ( sqrt(r1), 2*pi*r2 )
  (x,y) = (r*cos(theta),r*sin(theta))
Return
  x,y
```

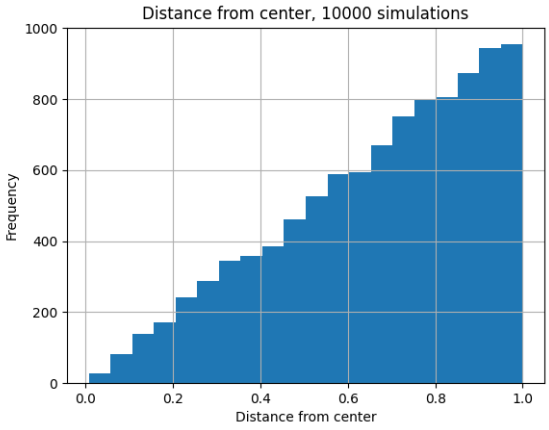
We then computed the distance d to the center, and added this to a running sum $dsum$. After n simulations, we estimated the average distance as $\frac{dsum}{n}$.

Now we would like to look more closely at this problem. In particular, we would like to see the probability of the distance values $0 \leq d \leq 1$. We would also like to answer the question: for any value $0 \leq \delta \leq 1$, what is the probability that, for a given simulation, the value of d we observe is less than δ?

In our previous simulation, we simply averaged the values of d to get a mean value, which turned out to be $\frac{2}{3}$. Now we want a more detailed view of how the simulated values spread out to either side of this average. We can do that by plotting a histogram of the data. Here, we call a function that simulates the fly landing `n` times, returning a vector `d` of every resulting distance value. The `matplotlib` function `hist()` will make a histogram of this data, using a given number of bins.

```
d = fly_simulation ( n )
plt.hist ( d, bins = 20 )
```

The resulting image is:



A histogram of 10,000 fly simulations.

This plot tells us something interesting. The behavior of d seems to fall along a straight line, beginning with zero flies at zero distance, and roughly 1,000 flies at a distance between 0.975 and 1.000. This suggests that there is a simple formula for the probability that a fly's distance is exactly some value d. In fact, because we are working with a continuous variable, we have to speak of a *probability density function* or PDF. If we have a function $pdf(d)$, then the probability that d falls in the interval $[d_1, d_2]$ is $pdf(d_2) - pdf(d_1)$.

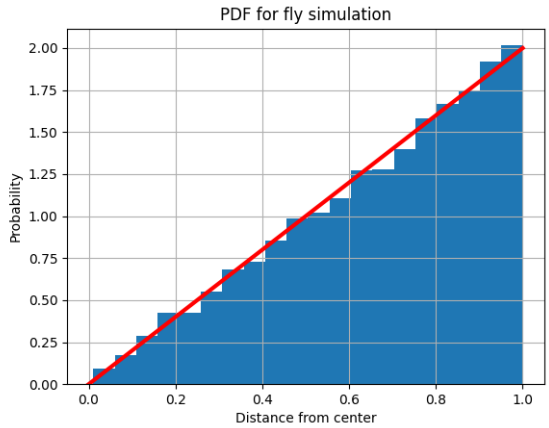
The shape of our histogram suggests the shape of the corresponding PDF. In fact, our pdf will be

$$pdf(d) = 2 \cdot d$$

which is 0 for $d = 0$, rises linearly over the interval $0 \leq d \leq 1$, and has the normalization property $\int_0^1 pdf(d) = 1$.

Because PDF's are such an important graphics tool, the `hist()` command includes an option that allows us to convert a histogram into an approximate PDF, by specifying the `density=True` option. Let us replot our data this way, and add a plot of the line defining the exact PDF:

```
d = fly_simulation ( n )
plt.hist ( d, bins = 20, density = True )
d2 = np.linspace ( 0.0, 1.0, 21 )
pdf = 2.0 * d2
plt.plot ( d2, pdf, 'r-' )
```



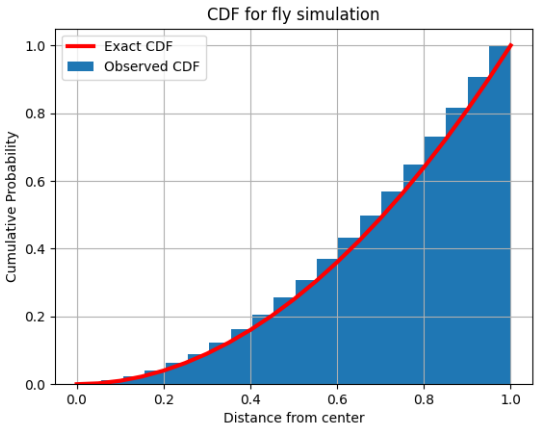
A PDF histogram of 10,000 fly simulations.

Notice that the scale along the y axis has changed from "frequency" to "probability".

Now a related question asks for the probability that the random variable d is less than or equal to some upper limit. For instance, suppose we have a fly zapper at the center of the plate, which will kill any flies with a radius of 0.5. What is the probabilty that a random fly will be killed? To answer this question, we want a *cumulative density function*, or CDF. For our problem, the value $cdf(dmax)$ is the probability that a random fly will land at a distance d that is no greater than $dmax$. For our question, then, we want to know the value $cdf(0.5)$.

Luckily, because the CDF is also an important function, `hist()` command includes another option that allows us to convert a histogram into an approximate CDF, by specifying both `density=True` and `cumulative=True`. As it turns out, the CDF function is the integral of the PDF function. So we also know the exact formula for our fly problem, namely, $cdf(d) = x^2$. So now let's plot our simulation data and the exact CDF:

```
d = fly_simulation ( n )
plt.hist ( d, bins = 20, density = True, cumulative = True )
d2 = np.linspace ( 0.0, 1.0, 21 )
cdf =  d2**2
plt.plot ( d2, cdf, 'r-' )
```



A CDF histogram of 10,000 fly simulations.

Just looking at the CDF plot, we can estimate $pdf(0.5) \approx 0.3$. The exact formula tells us that $pdf(0.5) = 0.25$, so things make some sense here.

12 Exercises

1. Given a set of n random values x, regarded as samples of some random probability distribution, the sample mean $\bar{x} \equiv \frac{1}{n}\sum_{i=0}^{i<n} x_i$, while the sample variance $\sigma^2(x) \equiv \frac{1}{n-1}\sum_{i=0}^{i<n}(x_i - \bar{x})^2$. The Python commands *np.mean()* and *np.var()* can be used to compute these quantities. Estimate $\bar{x}$ and $\sigma^2(x)$ for 1000 sample values of the logistic PDF.
2. Suppose that *two* flies land randomly on a 1 foot diameter plate. The exact PDF is $pdf(d) = \frac{1}{\pi} \cdot d \cdot (4\arccos(d/2) - d \cdot \sqrt{4 - d^2})$ Make a plot comparing your estimated PDF versus the exact formula.
3. Suppose that *three* flies land randomly on a 1 foot diameter plate. What is the average area a of the triangle their positions describe? Plot the estimated PDF and CDF for the value of a.

4. The Pareto distribution for parameters a and b, can be defined for $a \leq x$ as $cdf(x) = 1 - (\frac{a}{x})^b$. Derive the formulas for $pdf(x)$ and $icdf(x)$. Generate 10,000 sample values, and plot their histogram versus the PDF.
5. Estimate the probabilities of 1, 2 and 3 sixes for the Newton-Pepys dice problem. Report your estimates, and decide who was right. If you are a careful programmer, you can solve all three cases by adding one more line to the code, and slightly modifying two lines.
6. A country decides to try to increase the female population by controlling births. If a mother gives birth to a daughter, she is permitted to have another child next year. If she gives birth to a boy, she is not allowed to have any more children. Thus, in this country, there will be never be a family with more than one boy, but there will be families with three, six, even ten daughters. To what extent will this affect the ratio of boys and girls? Consider 1,000 families, and require each family to continue having a child every year until a boy is born. Compute the final overall ratio of boys to girls.
7. At the seaside, there is a pier that is 20 yards long. One end of the pier is the beach, and on the other is a sudden drop into the water. There is a bench every yard. A very sleepy sailor wakes up on the bench at the 10 yard line. The sailor staggers randomly to a neighboring bench and takes another nap. Every hour, the sailor wakes up and again staggers to a neighboring bench. If the sailor keeps doing this, he will fall off the end of the pier into the ocean, or else reach the beach and be safe. On average, how far from the 10 yard line is the sailor after 10, 20, ..., 100 hours?
8. N people play a game of odd-man-out. Each person tosses a coin. If one person's coin doesn't match all the others, that person is the odd man out, and the game ends. Otherwise, another round is played. Estimate the number of rounds played for values $3 \leq N \leq 10$.
9. 25 people attend a party. Use simulation to estimate the probabilty that at least two of these people have the same birthday.
10. Estimate the probability of being dealt a straight in poker.
11. A *flush* in poker is a hand in which all 5 cards have the same suit. Using our numbering scheme, for instance, the cards with index 14, 18, 19, 22, 25 would represent a flush, consisting of 5 cards of the club suit. Use simulation to estimate the probability of being dealt a flush in poker.