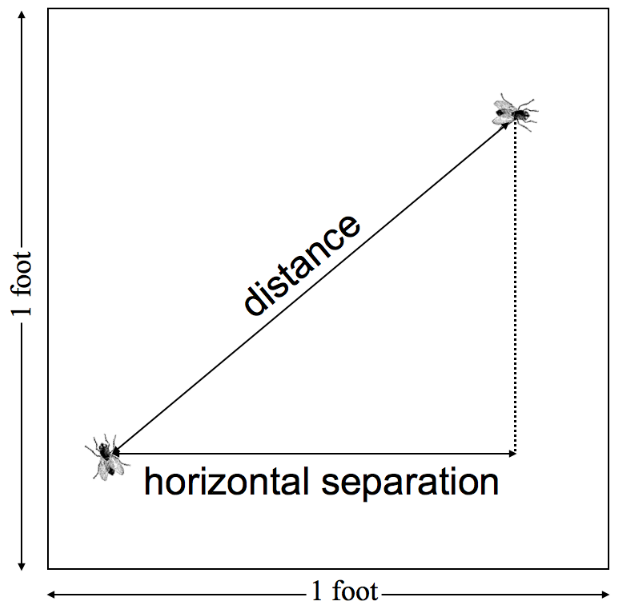
Python Demonstration

MATH1900: Machine Learning

Location: http://people.sc.fsu.edu/∼jburkardt/classes/ml_2019/python_demo/python_demo.pdf



"Two flies land at random on a square piece of paper. On average, how far apart are they?"

Python Demo

How can we use a programming language like Python to:

- *create a model of a mathematical problem;*
- *carry out a procedure to solve the problem;*
- *graphically illustrate the solution.*

Python is widely used for machine learning;

- it can be used interactively;
- a user can create new Python commands that extend the language;
- libraries are available for numerics, scientific computing, graphics;
- many machine learning packages have been built using Python.

The problem we will consider involves estimating the average distance of two flies who randomly settle on a square piece of paper. This particular problem can be solved analytically, but we are interested in how Python can simulate the problem, estimate the answer, and produce convincing tables and plots. We are studying a random process, and trying to say something (the average) about a huge number of cases.

Let's start by trying to simulate a single case.

1 One simulation

Let's assume the paper is one foot on each side. Then the position of any point on the square is simply a pair (x, y) of numbers between 0 and 1. We need a way to:

- pick a pair of numbers `x` and `y` at random, and pack them into a point `p`:
- compute the distance between two points `p1` and `p2`;

The Python library **numpy** contains many useful functions, which are organized into collections called modules.
The module **numpy.random** includes the useful function `rand`:

- `rand()` returns a single random number;
- `rand( n )` returns n random numbers;
- `p=rand(2)` returns 2 random numbers as an object named "p";

To simulate two random flies:

```
import numpy as np
p1 = np.random.rand(2)
print ( p1 )
p2 = np.random.rand(2)
print ( p2 )
```

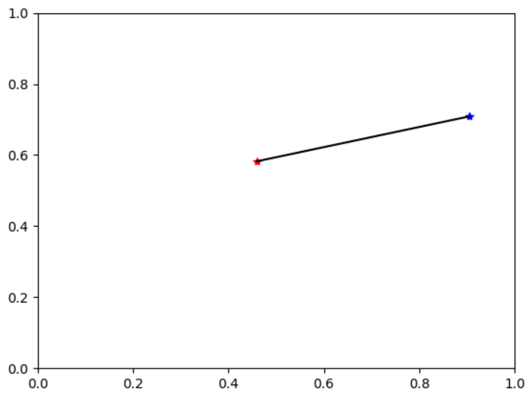
The module **numpy.linalg** includes the function `norm`:

- `norm(p)` returns the Euclidean length of the vector `p`;
- `norm(p1-p2)` returns the Euclidean distance between points `p1` and `p2`;

```
import numpy as np
p1 = np.random.rand(2)
p2 = np.random.rand(2)
d = np.linalg.norm ( p1 - p2 )
print ( d )
```

We can view the pair of flies using the **matplotlib** library:

```
import numpy as np
p1 = np.random.rand(2)
p2 = mp.random.rand(2)
import matplotlib.pyplot as plt
plt.plot ( p1[0], p1[1], 'r*' )
plt.plot ( p2[0], p2[1], 'b*' )
plt.plot ( [p1[0],p2[0]], [p1[1],p2[1]], 'k-' )
plt.axis ( [0,1,0,1] )
plt.show ( )
```



2 Many Simulations, Averaged

The average distance between random flies can be estimated by simulating many examples. In Python, the *for* loop lets us repeat an operation many times. We can initialize a variable `average` to zero, loop over `n` simulations, adding each distance to `average`, and then dividing by `n`:

```python
import numpy as np
average = 0.0
n = 10
for test in range ( 0, n ):
  p1 = np.random.rand(2)
  p2 = np.random.rand(2)
  d = np.linalg.norm ( p1 - p2 )
  average = average + d

average = average / n
print ( average )
```

If we all run this program, using `n=10`, how much variation do we see in our answers? Should that make us worry? Let's try `n = 100` and compare answers. What about `n = 1000`?

How large a multiple of 10 do we need before everyone's answer agrees to three decimal places?

3 Using Arrays

Rather than working with one case at a time, we could instead generate all the cases at once. In that case, we will be working with arrays. For instance, `p1` will be a $2 \times n$ array, where each column represents a separate case.

The `norm()` function must be told to separately compute the norms of each column of the array. Columns count as "axis 0" in Python arrays.

```python
n = 10
p1 = np.random.rand(2,n)
p2 = np.random.rand(2,n)
d = np.linalg.norm ( p1 - p2, axis = 0 )
average = np.mean ( d )
print ( average )
```

The advantages of writing a calculation in the vector version are that the code is somewhat more compact, it is easier to see that the computation could be done in parallel, and in particular, Python is able to compute many vector operations very efficiently.

4 Timing Comparison

Let's compare the scalar and vector codes when `n` is large.
First, let's rewrite our scalar calculation as a file, which we will call *simulate_many.py*, with the value of `n` an input parameter.

```
import numpy as np
def simulate_many ( n ):
  average = 0.0
  for test in range ( 0, n ):
    p1 = np.random.rand(2)
    p2 = np.random.rand(2)
    d = np.linalg.norm ( p1 - p2 )
    average = average + d

  average = average / n
  return average
```

Doing this means we have created our own Python command, which can be executed with a single line:

```
  average = simulate_many ( n )
```

We can similarly create *simulate_vector.py.* Now we could compare the speed of the two methods by simply waiting for each to complete, or using a stopwatch, but instead, we will use the Python command `time`.

In order to increase `n` from 10 to 100 to ... to 1,000,000, we use the exponentiation operator `**`.

In order to print three things and have them all show up on one line, we can add to the `print()` statement the argument `end=""`.

```
  from time import time
  for logn in range ( 1 : 7 ):
    n = 10 ** logn
    print ( n, end = "" )
    t = time ( )
    simulate_many ( n )
    t = time - t
    print ( t, end = "" )
    t = time ( )
    simulate_vector ( n )
    t = time ( ) - t
    print ( t )
```

We know that `simulate_many()` and `simulate_vector()` are computing the same things. When working with large sets of data, however, it can be a big time advantage to process all the data together, rather than doing them one at a time!

5 A convergence table

The average is an average over an infinite number of cases. We are estimating that average using `n` cases. How do we convince ourselves that the estimate is a good approximation? One way is to repeat the calculation for larger values of `n` and make a table. If the estimates seem to converge, then this suggests that our many choices of `n` seem to point towards a common answer *although we can't be sure this answer is right!*.

```
average_old = 0.0
for logn in range ( 1 : 7 ):
  n = 10 ** logn
  average = simulate_vector ( n )
  print ( n, ' ', average, ' ', average - average_old )
  average_old = average
```

Our results suggest that even a value of `n=1,000,000` only gives us confidence in the first two decimals of our estimate.

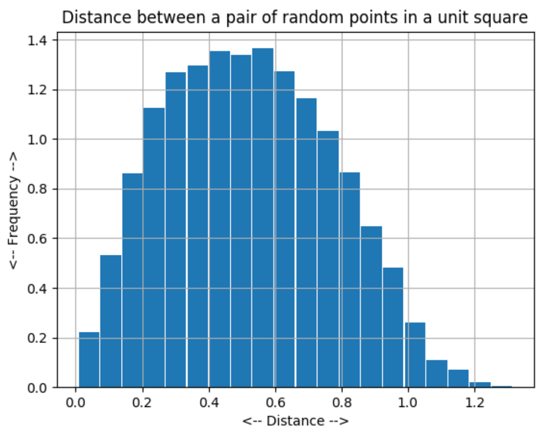
6 A histogram of the results

In the problem we are thinking about, possible distances d range between 0 and $\sqrt{(}2)$. Some distances are more likely than others, as described by a probability density function. If we choose `n` large enough, we can make a histogram of our results that will suggest what the true PDF looks like.
The basic command is simply

```
plt.hist ( d )
```

but I want to specify the number of bins to be 20, I want the bars to be slightly separated, and I want the vertical y axis to be scaled like a PDF, rather than a frequency count.

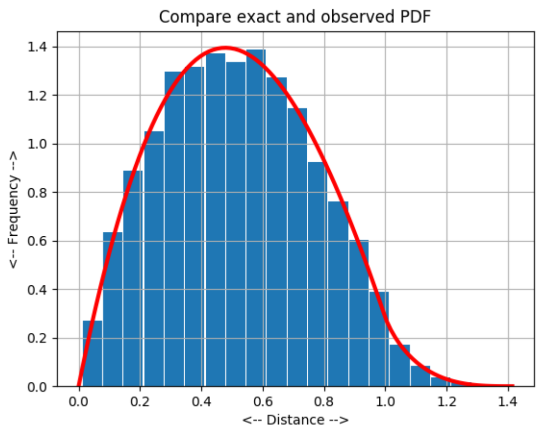
```
import matplotlib.pyplot as plt
import numpy as np
n = 10000
p1 = np.random.rand ( 2, n )
p2 = np.random.rand ( 2, n )
d = np.linalg.norm ( p1 - p2, axis = 0 )

plt.hist ( d, bins = 20, rwidth = 0.95, density = True )
plt.show ( )
```



7 Histogram versus PDF

For this particular problem, there happens to be an exact formula for the PDF. We can plot it together with the histogram, to see how we did. The formula for the exact PDF is a little messy, so I will skip the programming details:

$$\mathrm{pdf}(d) = \begin{cases} 2d(d^2 - 4d + \pi) & \text{if } d <= 1.0 \\ 2d(4\sqrt{d^2 - 1.0} - (d^2 + 2 - \pi) - 4\arctan(\sqrt{d^2 - 1})) & \text{if } 1.0 < d \end{cases}$$


Compare exact and observed PDF
<-- Frequency -->
<-- Distance -->