# python-modeling-kinser

## Table of Contents



# 1. The Monte Carlo Method


- Statistical technique used to solve mathematical problems through random sampling and simulation.
- Named after the famous Monte Carlo Casino in Monaco
- Repeatedly sample random values for uncertain variables then compute the results of a mathematical model or system with these variables.
- Iterations generate a distribution of outcomes, which can be analyzed to gain insights in the behavior of the system modeled.
- Useful for solving problems that may be difficult or impossible to solve analytically, for example integrals in high-dimensional spaces.
- Books and articles:
    1. Sobol, 1994 - Primer
    2. Kahn, 1954 - Applications
    3. Senova et al, 2023 - Risk assessment
    4. Mazzola, 2022 - Quantum computing
    5. Li, 2023 - Monte Carlo Approximation Methods

## 1.1. Overview

You will learn:

- How to generate random vectors
- How to check if a function was loaded
- How to create a random vector
- How to set a random seed
- How to create a random matrix
- How to simulate rolling dice
- How to determine the area below a line
- How to estimate the area under an arbitrary curve
- How to estimate the area in a square
- How to estimate $\pi$

## 1.2. Random vectors

If you generate a random point in a three-dimensional space, the $x$, $y$ and $z$ components of the vector would need to be random numbers.

The `rand` function in the `numpy.random` module returns multiple random values in an array. Let's load it first:

```
from numpy.random import rand
```

## 1.3. Check if a function was loaded

How can you check if a function was loaded (without trying to run it and generating an error? In R, there is a convenient function for that (`search`) but in Python, you have to write one.

The `dir` function lists all methods and attributes of a function. For example, if `dir(rand)` is not `None`, the function is loaded.

Test that with `numpy.random.rand`:

```
print(dir(rand))
```

Write a function `loaded` that accepts a function name and checks if it is loaded using `dir`. Test it with:

1. with `numpy.random.rand`.
2. with `exit` from the `sys` library.
3. with the `datetime` module from the `datetime` library.
4. show the current time with `now` from `datetime.datetime`.

```
def loaded(function):
    '''Check if function was loaded.

    function: return __name__ attribute
    '''
    if dir(function) != None:
        return print(f'{function.__name__} loaded')
    return print(f'{function.__name__} not loaded')

from numpy.random import rand
loaded(rand)

from sys import exit
loaded(exit)

from datetime import datetime
loaded(datetime)

# show the current time
print(datetime.now())
```

This is easier than searching through a list of all functions:

```
[print(f'{name} {value}') for name, value in globals().items()];
```

## 1.4. Create a random vector

Back to random numbers! Load `numpy` as `np` and then create a vector `vec` of three random numbers using `rand`:

```
import numpy as np
vec = np.random.rand(3)
print(vec)
```

Notice how you get different numbers each time you run the code block.

## 1.5. Repeating random numbers

To get the same result, you can set the `seed`:

```
np.random.seed(11212023)
print(vec)
```

Wrap this code into a function `Seed`:

1. when called without an argument or with `0`, set `seed` to `None`.
2. when called with a non-zero argument, set `seed` to that number.

```
def Seed(num=0):
    '''Set random seed
    num: random seed (default 0 or None)
    '''
    if num == 0:
        np.random.seed(None)
    else:
        np.random.seed(num)
```

Check that `seed` is loaded:

```
loaded(Seed)
```

Why can you be sure that `loaded` checked for your function `Seed` and not for the function `seed` from the `numpy` package?

> The `seed` function was implicitly loaded with all of `numpy` with the command `import numpy as np` and needs to be addressed as `np.random.seed`, and our function is `Seed` and not `seed`.

Test your function `seed`.

```
# set seed to None
Seed()

# create a vector of three random numbers and print it thrice
[print(np.random.rand(3)) for i in range(3)]

# set seed to today's date
Seed(1)
```

```
# create a vector of three random numbers and print it thrice
[print(np.random.rand(3)) for i in range(3)]
```

## 1.6. Create a large number of random vectors

To create a large number of random vectors, create a matrix of random values: the function `ranf` accepts a single argument, a tuple of two integers with the size of the matrix `mat`:

```
# create 100 x 3 random matrix
mat = np.random.ranf((100,3))

# print the first three rows of the matrix
print(mat[0:3])  # same as mat[0:4,] or mat[0:4,:]
```

To limit the number of decimals printed (not generated), set the `numpy` function `set_printoptions` to 3, then print the first four rows of `mat`:

```
np.set_printoptions(precision=3)
print(mat[0:4])
```

`numpy` statistics are vectorized: compute the average across columns:

```
print(mat.mean(0))
```

The average for the normal distribution of a sequence of pseudorandom numbers between 0 and 1 should be close to 0.5.

## 1.7. Rolling dice

Rolling a six-sided die is a common example for random number generators. To generate random integers, we can use `randint` from `numpy.random`. Its first two arguments are the lower bound (included) and the upper bound (excluded), its 3rd argument is number of draws:

```
import numpy as np

print(np.random.randint(1,7))  # roll die once
print(np.random.randint(1,7,10)) # roll die 10 times
```

By sending a tuple to the third argument, the result is a random matrix:

```
print(np.random.randint(1,7,(5,2))) # 5 x 2 random matrix of values in {1-6}
```

Challenge: roll 5 dice 10,000,000 times and store the result in `mat`:

```
N = int(1e7)
mat = np.random.randint(1,7,(N,5))
print(mat[0:4]) # the first four draws with 5 fair dice
```

Find all the cases in which the 5 dice have the same values:

```
# create a new N x 4 matrix of zeros
minus = np.zeros((N,4))

# the value in minux[m,n] is the abs value of mat[m,0]-mat[m,n+1], or
# the difference of one row value to the first value in the row
for i in range(4):
    minus[:,i] = abs(mat[:,0]-mat[:,i+1])

# sum horizontally across minus columns - adds has N elements
# if a row n in mat has the same values, then adds[n] == 0
adds = minus.sum(1)

# count the number of times that 0 appears in adds
ct = (adds==0).sum()

# print count
print(f'Number of identical dice rolls in {N} rolls = {ct}')
```

Mathematically, the probability that all dice show the same number in one roll is N * $(1/6)^4$:

```
N = 10000000  # Number of rolls
probability_same_number = (1/6)**4

# Calculating the expected count
expected_count = N * probability_same_number
print(expected_count)
```

## 1.8. The Monte Carlo Method recipe

The Monte Carlo method generates random data and receives results from the interaction of that data.

A simple example: generate points in a defined space and determine which side of a decision surface the points reside.

We're going to discuss several examples of increasing complexity:

- Horizontal barrier (1d)
- Slanted barrier (1d)
- Simple curve (1d)
- Complex curve (1d)
- Square (2d)
- Circle (2d)

## 1.9. Estimating the area below a horizontal barrier

For this example, the line is determined by y = 0.5. We generate random points and determine if they are above or below a decision line.

Since the points are evenly distributed, the space is sampled uniformly, and the ratio of the number of points below the line to the total number of points is similar to the ratio of the area below the line to the total area.

Begin by creating a 100 x 2 random matrix (two vectors of 100 elements each, one for x one for y coordinates) with `ranf` from `numpy.random` and print the first four (x,y) pairs:

```
# import numpy as np
import numpy as np

# create 100 x 2 random matrix
vecs = np.random.ranf((100,2))
print(vecs[0:4])
```

Compare the y values of these vectors to 0.5 and store the resulting Boolean vector in a variable `above`:

```
above = vecs[:,1] > 0.5
print(above[0:4])
```

The function `np.nonzero` returns the non-zero values of an array. `ndx` contains the indices of all non-zero (`True`) points above the line.

```
ndx = above.nonzero()[0]
print(ndx[0:4])
```

We create a scatter plot of these points:

```
# import graphics module
import matplotlib.pyplot as plt

# plot points above the line
plt.clf()
plt.scatter(x=vecs[ndx,0],y=vecs[ndx,1])
plt.savefig('img/above.png')
```

Switch the Boolean values in `above`, making `ndx` the indices of the points below the line, and print those points with a marker:

```
plt.clf()
ndx = above.nonzero()[0]
plt.scatter(x=vecs[ndx,0],y=vecs[ndx,1])
ndx = (1 - above).nonzero()[0]
plt.scatter(x=vecs[ndx,0],y=vecs[ndx,1],marker='+')
plt.axhline(y=0.5, color='red', linestyle="dashed") # Add horizontal line at y=0.5
plt.xlim(0,1)
plt.ylim(0,1)
plt.title("Estimate the area below a horizontal barrier at y=0.5")
plt.savefig('img/above_and_below.png')
```

## 1.10. Estimating the area below a slanted barrier

To divide the random points by a slanted line, notice that the dividing line (and our "decision surface") is defined by $y = mx + b$ where $m$ is the `slope` and $b$ is the `intercept`.

As before, we create 100 random vectors in the plane, set the `slope` and `intercept` values:

```
slope = -1.3
intercept =  0.8
vecs = np.random.ranf((100,2))
```

Plot the points (with `scatter(x,y)` and the line (with `plot(x,y)`):

```
# import graphics and numeric libraries
import matplotlib.pyplot as plt
import numpy as np

# clear graphics
plt.clf()

# plot random vectors
plt.scatter(x=vecs[:,0],y=vecs[:,1])

# create array x of 2 values in [0,1]
x = np.linspace(0,1,2)

# generate line with slope and intercept
y = x * slope + intercept

# plot line
plt.plot(x,y,color="red",linestyle="dashed")

# fix display limits for viewing
plt.ylim(-0.1,1.1)
plt.savefig('img/slanted.png')
```

To determine if an (x,y) point is above the line, the y value of the point is compared with the corresponding value in yline. Convert the resulting Boolean vector into numbers with `nonzero`:

```
# create line taking the x array from the random vectors
yline = vecs[:,0] * slope + intercept

# check if y value is above the line and store in ndx index vector
ndx = (vecs[:,1] > yline).nonzero()[0]
```

Plot the resulting points:

```
plt.clf()
plt.scatter(vecs[ndx,0],vecs[ndx,1])
plt.savefig('img/yline_above.png')
```

Do the same for the points below the line: create Boolean vector, convert it to indices and store it in `ndx`, then plot the result and mark the points with a `+`:

```
# check if y value is below the line and store in ndx index vector
ndx = (vecs[:,1] < yline).nonzero()[0]

plt.clf()
plt.scatter(vecs[ndx,0],vecs[ndx,1],marker='+')
plt.savefig('img/yline_below.png')
```

Put the code blocks together and plot the line using y defined earlier to show the decision surface:

```
import matplotlib.pyplot as plt
import numpy as np

# define the decision surface
```

```
yline = vecs[:,0] * slope + intercept

plt.clf()

# points above the line
ndx = (vecs[:,1] > yline).nonzero()[0]
plt.scatter(vecs[ndx,0],vecs[ndx,1])

# points below the line
ndx = (vecs[:,1] < yline).nonzero()[0]
plt.scatter(vecs[ndx,0],vecs[ndx,1], marker='+')

# draw the line
plt.plot(x,y,color="green",linestyle="dashed")

# fix the display limits
plt.xlim(0,1)
plt.ylim(0,1)

plt.savefig('img/yline_above_below.png')
```

```
plt.clf()
plt.plot(x,y)
plt.ylim(0,1)
plt.savefig('img/line.png')
```

## 1.11. Integration under a curve: simple function

### 1.11.1. A simple trigonometric function

Consider the following function:

$$y = 3x + 2\sin(25x) + 2 \tag{1}$$

The function is easy to integrate but our method would also work for other functions that are difficult to integrate analytically (see below):

$$\int ((3/2)x + \sin(25x))\, 25x = \frac{a}{2}x^2 - \frac{1}{25}\cos(25x) \tag{2}$$

Define the function in Python: let's keep the parameters separate for now - this would make it much easier to modify the curve's behavior:

```
# Defining the function:
def f(x):
    # compute y from x
    y = 3*x + 2 * np.sin(25*x) + 2
    # create plot label
    label = '3x + 2sin(25x) + 2'
    # return y and plot label
    return y, label
```

Plot the function:

```
import matplotlib.pyplot as plt
```

```
# Define the range for x
x = np.linspace(0, 1, 100)  # Adjusted to the x-axis of the image
y, plot_label = f(x)

# Create the plot
plt.clf()
plt.figure
plt.plot(x, y, label=plot_label)
plt.title('Plot of the given function')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.savefig('img/function.png')
```

### 1.11.2. Monte Carlo integration

To apply Monte Carlo, generate random points, pepper the plane with them, check which points are below the line, and sum them up.

Generate 100 random vectors: because we know the range of y values, let's scale them up:

```
# generate 100 random (x,y) points
xpts = np.random.ranf((100,2))

# y values are in the range 0 to 6.5: scale them up
xpts[:,1] *= 6.5
```

Plot the random vectors with the curve:

```
plt.clf()
plt.scatter(xpts[:,0],xpts[:,1])
plt.plot(x, y, color="red",linestyle="dashed")
plt.grid(True)
plt.savefig('img/function1.png')
```

Now, compute the function for all x and generate a Boolean vector from the points below the curve:

```
yline, plot_label = f(xpts[:,0])
area = xpts[:,1] < yline
```

Summing over the resulting vector gives the total number of points below the curve:

```
print(area.sum())
```

Around 50% out of 1000 points are below the line. The average of the function is a straight line that cuts the figure roughly in half.

### 1.11.3. Plotting challenge

Print the points above the line in color, and the points below the line with the marker '+'.

Solution:

The random vectors are stored in `xpts`. To print vector subsets, we have used different x index subsets obtained with the `nonzero` function, which returns the indices of the array elements that are non-zero.

The elements that are non-zero are equivalent to the `True` elements if we check all points for the condition above/below f(x).

In code:

```
points_below = (xpts[:,1] < yline)
x_below = points_below.nonzero()[0]
```

- `x_above` is the vector of indices of points below the line.
- `xpts[:,1]` is the vector of random y values
- `yline` is the vector of function values on the curve f(x)
- `nonzero` extracts `True` values from the Boolean `points_below`
- `[0]` extracs the x values from the result as an array

Same for the points below: we use the trick of turning the Boolean into its inverse for the indices of the remaining points:

```
x_above = (1 - points_below).nonzero()[0]
```

Now we have all we need to plot both subsets:

```
plt.clf()
plt.scatter(xpts[x_above,0],xpts[x_above,1])
plt.scatter(xpts[x_below,0],xpts[x_below,1],marker='+')
plt.plot(x, y, color="red",linestyle="dashed")
plt.grid(True)
plt.savefig('img/function2.png')
```

## 1.12. Integration with `scipy` and by hand

Integration with `scipy.integrate.quad`:

```
from scipy.integrate import quad

# Define the function to integrate
def integrand(x):
    return 3*x + 2 * np.sin(25*x) + 2

# Calculate the area under the curve from x=0 to x=1
result, _ = quad(integrand, 0, 1)

print(result)
```

That's about 50% of the total area 7 x 1 = 7.

Manually: The integral of the function $3x + 2\sin(25x) + 2$ is: $$

$$\int (3x + 2\sin(25x) + 2)\,dx = \frac{3}{2}x^2 - \frac{2}{25}\cos(25x) + 2x + C$$

$$ where $C$ is the constant of integration.

To find the area under the curve from $x = 0$ to $x = 1$, evaluate this antiderivative at 1 and 0: $$

$$\left[\frac{3}{2}x^2 - \frac{2}{25}\cos(25x) + 2x\right]_0^1 = \left(\frac{3}{2} \times 1^2 - \frac{2}{25}\cos(25 \times 1) + 2 \times 1\right) - \left(\frac{3}{2} \times 0^2 - \frac{2}{25}\cos(25 \times 0) + 2 \times 0\right)$$
$$= \frac{3}{2} - \frac{2}{25}\cos(25) + 2 + \frac{2}{25} \approx 3.5$$

$$

## 1.13. Integration under a curve: complex function

### 1.13.1. A complex function

Analytic integration of this function to establish the area under the curve is not straightforward due to the composition of trigonometric functions and their powers. Numeric integration is our best bet:

$$y = \left(\sin\left(\left(\cos\left((x + 10) \times 7\right)\right)^4 \times 8\right)\right)^2 + \left(\cos\left((x + 10) \times 70\right)\right)^4 + \exp\left(\frac{x + 0.5}{0.9}\right) \tag{3}$$

Let's define the function:

```
# Define the function to plot
def g(x):
    label = 'y = sin(8cos(7(x + 10))^4)^2 + 70cos(x+10)^4+exp((x + 0.5) / 0.9)'
    y = (np.sin((np.cos((x+10)*7))**4*8))**2+(np.cos((x+10)*70))**4+np.exp((x+0.5)/0.9)
    return y, label
```

To plot, we need to specify a range of x values:

```
# Create an array of 400 x values from 0 to 10
x = np.linspace(0, 10, 400)

# Generate the y values by applying the function to the x values
y, plot_label = g(x)

# Plot the function
plt.clf()
plt.plot(x, y, label=plot_label)
plt.savefig('img/kinserfunc1.png')
```

You can see from the result that the oscillations that correspond to the trigonometric functions are not visible because the exponential term dominates. To see these oscillations, reduce the display window to $x \in [0, 1]$:

```
# Create an array of 400 x values from 0 to 1
x = np.linspace(0, 1, 400)

# Generate the y values by applying the function to the x values
y, plot_label = g(x)

# Plot the function
plt.clf()
plt.figure(figsize=(12, 6))
plt.plot(x, y, label=plot_label)
```

```
plt.plot(x,3.5*x+2.2, color="green",linestyle="dashed", label='y = 3.5x+2.2')
plt.title('Plot of the given function')
plt.xlabel('x')
plt.ylabel('y')
plt.ylim(0,7)
plt.legend()
plt.grid(True)
plt.savefig('img/kinserfunc.png')
```

### 1.13.2. Monte Carlo integration

Other than using a different function, the code is the same as before:

1. Create 100 random vectors in [0,1] and scale y values according to the y range of the function.
2. Compute decision surface using random y values and the function.
3. Create a Boolean vector for the x values of points below the curve.
4. Sum up the `True` values of the Boolean vector to get the number of points below the curve.
5. Check your calculation by estimating the area under the curve as a percentage using averages.
6. Print points above and below the line together with the curve.

Solution:

```
# create random points
xpts = np.random.ranf((100,2))

# scale random y values
xpts[:,1] *= 6

# compute decision surface
yline, plot_label = g(xpts[:,0])

# select points below the curve
area = xpts[:,1] < yline

# compute total number of points
print(area.sum())
```

The result hovers around 66% which is in line with the area ratios when averaging the function as a line f(x) = ax + b for a=3.5 and b=2.2 (obtained manually, by trying different values).

Let's integrate the area below the line as before:

```
def f1(x):
    label = 'y = 3.5x+2.2'
    y = 3.5 * x + 2.2
    return y,label

vec1 = np.random.ranf((100,2))
vec1[:,1] *= 6
yline1, label1 = f1(vec1[:,0])
area1 = vec1[:,1] < yline1
print(area1.sum())
```

The results are comparable.

### 1.13.3. Iterate the Monte Carlo simulation

Instead of running this code manually, run N simulations of a function and store the result of each iteration in an array, then average it:

```
# Define the function to plot
def h(x):
    return (np.sin((np.cos((x+10)*7))**4*8))**2+(np.cos((x+10)*70))**4+np.exp((x+0.5)/0.9)

# define simulation function
def mc(N,M):
    # create empty array for storage
    result = np.zeros(N)
    for i in range(N):
        # create random points
        xpts = np.random.ranf((M,2))
        # scale random y values
        xpts[:,1] *= 6
        # compute decision surface
        yline = h(xpts[:,0])
        # select points below the curve
        area = xpts[:,1] < yline
        # compute total number of points
        result[i] += area.sum()
    # return resulting array
    return result
```

Run N = 100,000 simulations and print the average:

```
N = 10000 # number of iterations
M = 10000 # number of points per iteration
print(f'{100 * mc(N,M).mean()/M:.2f}% of the total area over {N} iterations.')
```

## 1.14. How to estimate the area of a square

We turn to 2-dimensional decision surfaces. Our target area is the lower left quadrant of a 1 x 1 square: it occupies 1/4 of the total environment, and we expect that one quarter of the total random values will land inside the target.

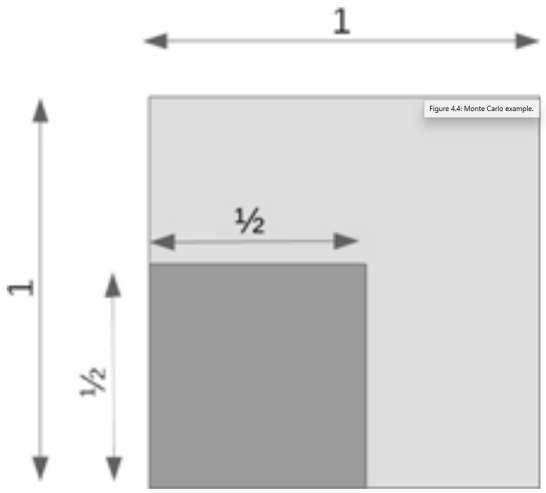


Figure 1: Source: Kinser (2021), p. 33.

The steps to solve the problem:

1. Generate 10,000 random points. The space is in [0,1] so you do not need to bias or offset the random values.

```
import numpy as np

# generate 10,000 random points in the plane
vecs = np.random.ranf((10000,2))

print(vecs[0:5])
```

2. Determine if a point is inside of the target area or not: $x > 0 \wedge x < 0.5 \wedge y > 0 \wedge y < 0.5$. Since the random values automatically meet the first and third condition, these reduce to: All pairs $(x, y)$ for which $x < 0.5 \wedge y < 0.5$ is `True`.

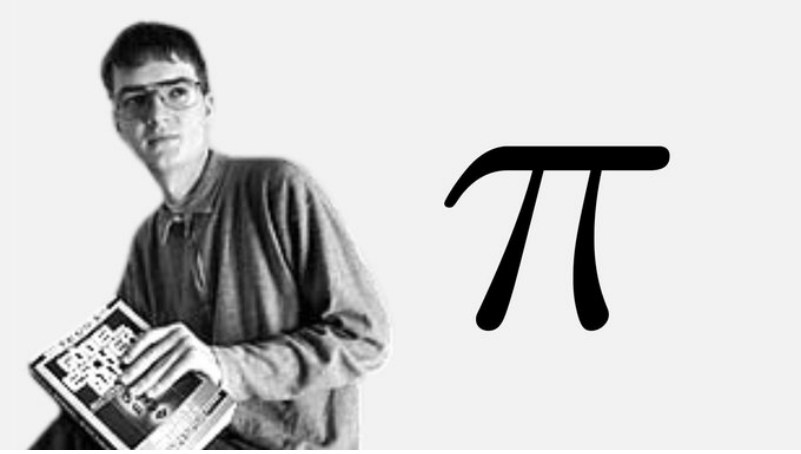
```
# store indices for which the condition is True in Boolean variables
xless = vecs[:,0] < 0.5
yless = vecs[:,1] < 0.5

# determine indices for which both conditions hold
both = xless * yless # 0 if either multiplicand is 0, and 1 otherwise
print(both[:5])

# Count number of points in target region by summing
print(both.sum())
```

   This is close to the expected value of 10,000/4 = 2,500 points.

## 1.15. How to estimate Pi

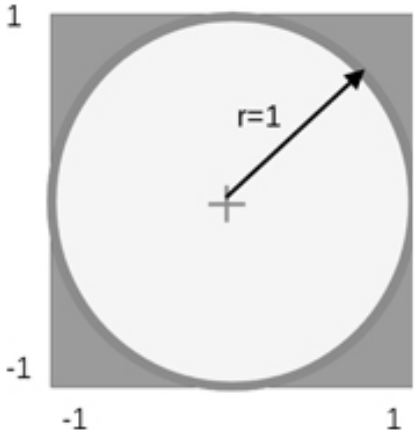
### 1.15.1. Fun fact

> "In 1998, 17-year-old Colin Percival calculated the 5 trillionth binary digit of π. This accomplishment was significant not only because it broke a record, but also because it marked the first time such calculations were distributed among 25 computers worldwide. The project, named PiHex, took 5 months of real time and a year and a half of computer time to complete. Percival, who graduated from high school in June 1998, had concurrently been attending Simon Fraser University in Canada since he was 13." @fermatslibrary (PiHex)



### 1.15.2. Environment to estimate $\pi$

Consider a circle inscribed of a square $(x \in [-1,1], y \in [-1,1])$




The radius of the circle is 1 and the length of the sides of the square is 2. We generate random locations inside of the square.

The ratio of the points inside the circle $N_c$ to the total number of points is the same as the ratio of the areas of the circle $A_c = \pir^2$ over the area of the square $A_s$.

For $r = 1$, therefore $A_c = \pi$, or $N_c/N = \pi/4$, and we obtain $\pi = 4\frac{N_c}{N}$.

For our decision surface, we use the distance $d = \sqrt{x^2 + y^2}$ to the center of the environment (0,0), therefore if \(d

The function $pi_mc$ generates the random points, `vecs`, computes the distances of each point (x,y) to the center, and evaluate the condition $d < 1$ and computes an estimate for $\pi$.

```
import numpy as np
def pi_mc(N,D=2):
    '''Return estimate for Pi

    N: number of random points in [-1,1]
    D: random matrix dimension (default D=2)
    '''
    # create random vectors in a square (x,y) in [-1,1]
    vecs = 2 * np.random.ranf((N,D)) - 1

    # compute distance of random points to center (0,0) sum(1) sums
    # the distances along axis 1 (for each x,y pair) the result is an
    # array (N,) which represents the sum of squares of each row of
    # vecs
    dists = np.sqrt((vecs**2).sum(1))

    # select points inside the circle and sum them up
    N_c = (dists < 1).sum()

    # compute estimate for Pi
    result = 4.0 * N_c / N

    # return result
    return result
```

Compute Pi with 1 mio random points:

```
N = 10000000
pi_est = pi_mc(N)
print(f'For {N} randnom points, PI is {pi_est}.')
print(f'A more accurate value is {np.pi}.')
```

Created: 2023-11-29 Wed 07:20