```
Out[34]: 0.5707963267948966
```

Using the formula directly:

```
In [35]: np.pi/2-1
```

```
Out[35]: 0.5707963267948966
```

4.1.7 And as you would expect all of these methods are giving us consistent results.

5 Random Walk: As a simple example of modeling a random process

5.1 We are going to make a random walk and try to answer basic questions like what is the expected distance from the staring point, path, ...

There are few ways of appraoching this problem: 1. **Functional approach**: Building the whole process as a pipeline of differnet functions, which is the approach we used so far. 2. **Object oriented approach**: Which is concepetually a very differnet approach but by doing this excersise we'll learn why this approach can be very useful for some problems.

Tip: Desiding what appraoch to take for a particular problem, depends on many factors. One of the easiest factor which is mainly independent of the problem is that whether you are going to reuse your codes again or adding differnet features to it later. If that's the case, generally speaking it is better to try to think about the problem and implement your code with the object oriented approach.

5.1.1 Like always let's start with the simplest case: 1-d random walk

```
In [36]: def random_walk_1d(n, step=1):
             """This is a function for making a 1-d random walk
             INPUT:
                   n (int): number of steps to take
                   step (float): lenght of each steps

             OUTPUT:
                   positions (numpy.array): an array of different positions during
                 the random walk

             """
             import random
             import numpy as np
             # making an array for putting all the information
             positions=np.zeros(n)

             # initial position
             x = positions[0]

             for i in range(1,n):
                 # choosing the random step to take
                 dx = random.choice([1,-1])*step
```

```
            x+=dx
            positions[i]=x
        return positions
```

5.1.2 Let's use our code above for finding a min(max) of a given function:

```
In [37]: def f(x):
             return x**2+2*x
```

We can find the minimum:
$\frac{df(x)}{dx} = 2x + 2$
Which has the solution of $x = -1$

```
In [38]: def min_finder(_f, xi=0, step=0.01, n=100):
             import random
             import numpy as np

             positions=np.zeros(n)
             x = xi
             for i in range(1,n):
                 stay = 1
                 while stay==1:
                     dx = random.choice([1,-1])*step
                     x_dummy=x+dx
                     if _f(x_dummy)<=_f(x):
                         positions[i]=x_dummy
                         x = x_dummy
                         stay = 0
             return positions
```

We got the same results with our greedy random walk algorithm:

```
In [39]: min_finder(f, n=100)[-1]
```

```
Out[39]: -0.9900000000000007
```

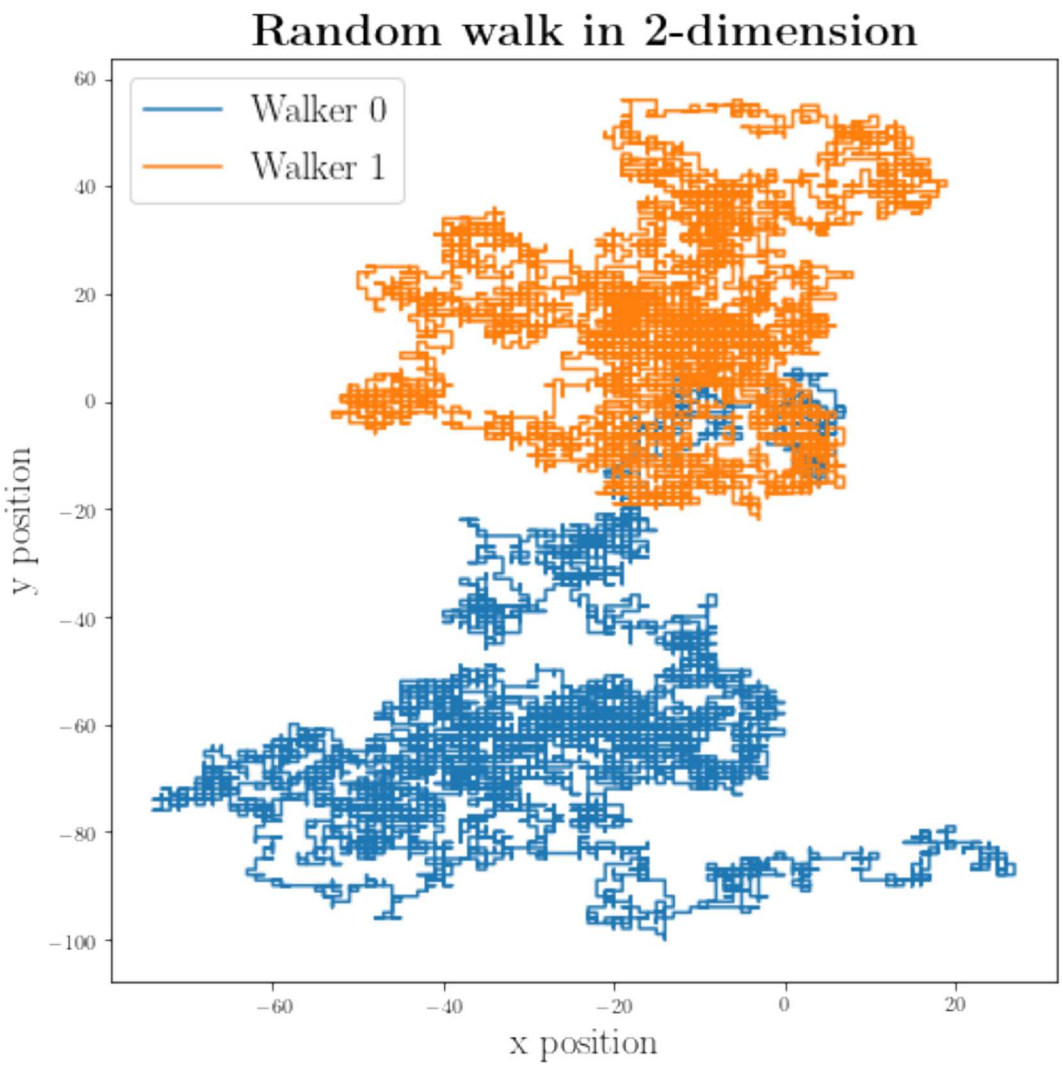
5.1.3 Now let's go to 2-d case: 2-d random walk

```
In [40]: def random_walk_2d(n):
             """2-d random walk function
             INPUT:
                  n (int): number of steps to take

             OUTPUT:
                  positions_dic (dic): A dictionary which contains am array of x and y values
                  the keys are "x" and "y"
             """
             x,y = 0, 0
             import random
```

```
    positions_dic={}
    positions_dic["x"]=np.zeros(n)
    positions_dic["y"]=np.zeros(n)
    for i in range(1, n):
        (dx, dy) = random.choice([(0,1), (0,-1), (1,0), (-1,0)])
        x,y=x+dx,y+dy
        positions_dic["x"][i]=x
        positions_dic["y"][i]=y
    return positions_dic
```

In [41]:
```
N = 10000

positions_walker_0=random_walk_2d(N)
positions_walker_1=random_walk_2d(N)
```

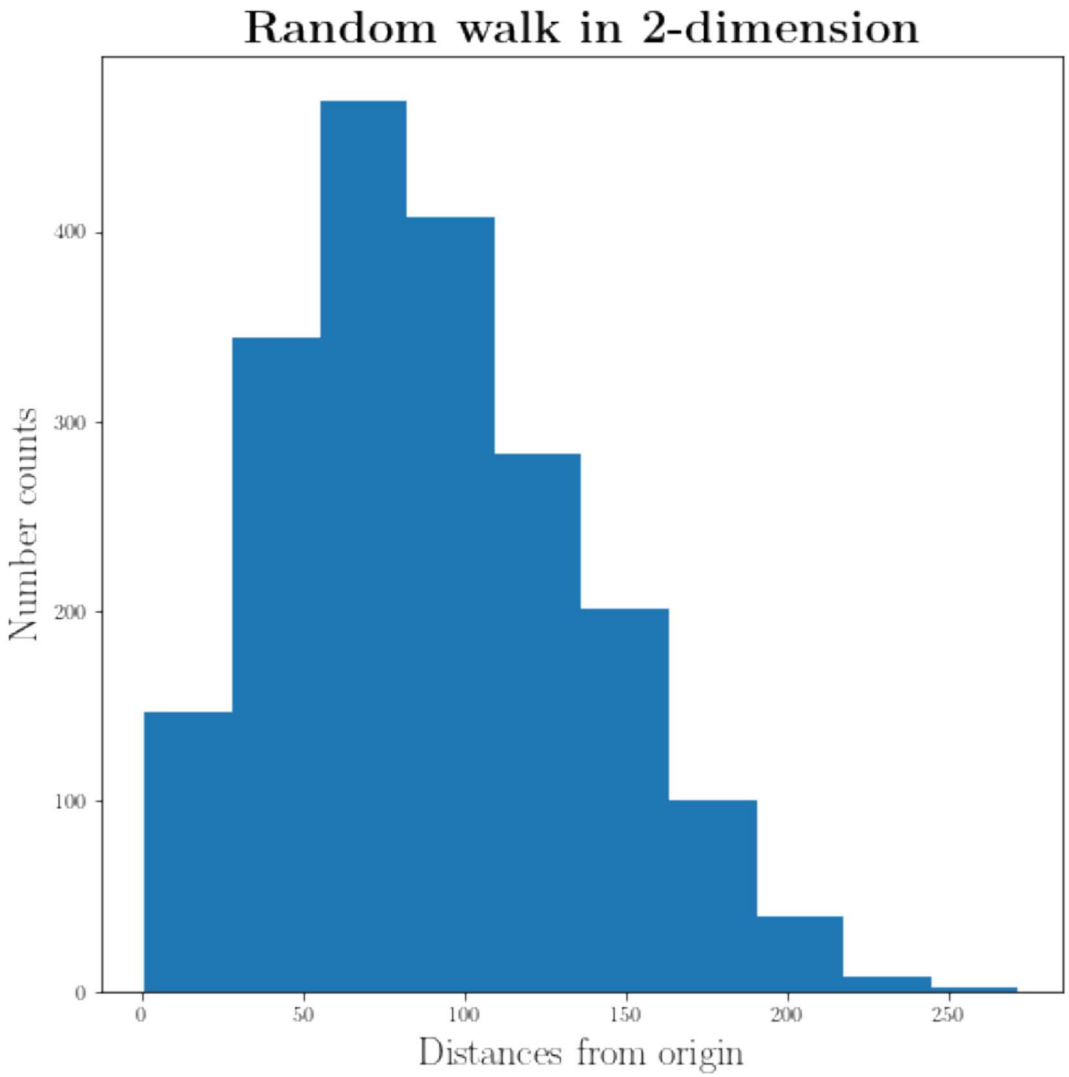
In [42]:
```
fig_2d_rw = plt.figure(figsize=(8,8))

plt.plot(positions_walker_0["x"], positions_walker_0["y"], label="Walker 0")
plt.plot(positions_walker_1["x"], positions_walker_1["y"], label="Walker 1")

plt.title(r"\textbf{Random walk in 2-dimension}", fontsize=22)

plt.xlabel(r"x position", fontsize=18)
plt.ylabel(r"y position", fontsize=18)

plt.legend(fontsize=18, markerscale=10)


plt.show()
```



Now that we have our random walk function we can run a simulation of N step random walk and record the distances from origin:

```
In [43]: Number_of_simulations = 2000
         distance=np.zeros(Number_of_simulations)

         Number_of_walks_in_each_simulation = 10000

         for i in range(Number_of_simulations):
             positions=random_walk_2d(Number_of_walks_in_each_simulation)
             distance[i]=np.sqrt(positions["x"][-1]**2+positions["y"][-1]**2)


In [44]: hist_rw = plt.figure(figsize=(8,8))
```

```
plt.hist(distance)

plt.title(r"\textbf{Random walk in 2-dimension}", fontsize=22)

plt.xlabel(r"Distances from origin", fontsize=18)
plt.ylabel(r"Number counts", fontsize=18)


plt.show()
```




```
In [45]: np.sqrt(Number_of_walks_in_each_simulation)

Out[45]: 100.0
```

The $\sqrt{N}$ in which N is number of steps is very close to the most likely value. Something to check and think about later.

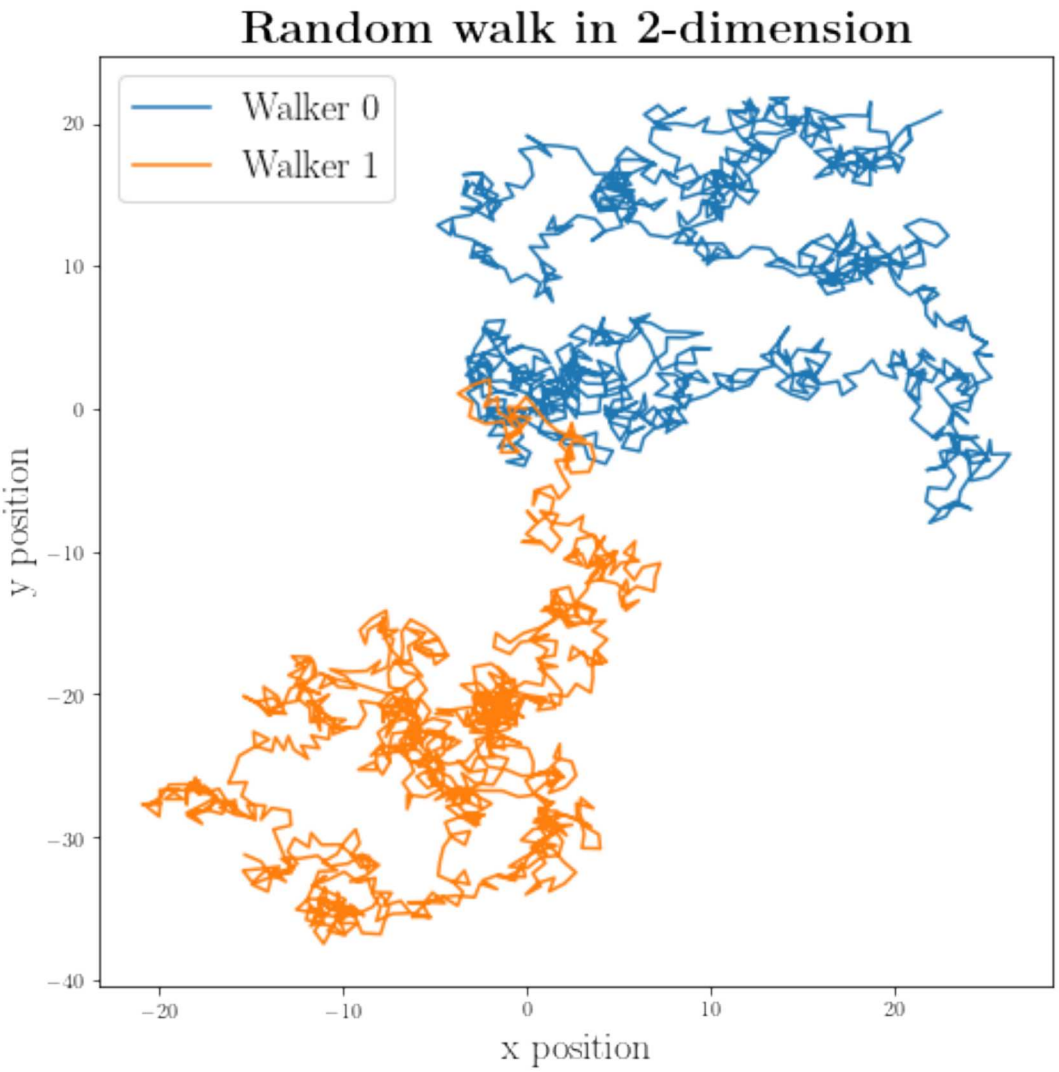
Here we are writing a function to simulate N simulations with n steps:

```
In [48]: def Simulate_walks(number_of_steps, number_of_simulations):
             simulation={}
             for i in range(number_of_simulations):
                 simulation[i]=random_walk_2d(number_of_steps)
             return simulation
```

5.1.4 Now let's make a random walker which can move along differnt angles.

```
In [54]: def random_walk_2d_degree_free(n):
             x,y = 0, 0
             degree=0

             import random
             import numpy as np

             positions_dic={}
             positions_dic["x"]=np.zeros(n)
             positions_dic["y"]=np.zeros(n)

             for i in range(1, n):
                 # Choose a degree in radian between [0, 2*pi] with 100000 choices for angles
                 degree = random.choice(np.linspace(0, 2*np.pi, 100000))
                 (dx, dy) = (np.cos(degree), np.sin(degree))
                 x,y=x+dx,y+dy
                 positions_dic["x"][i]=x
                 positions_dic["y"][i]=y
             return positions_dic

In [67]: positions_walker_deg_0 = random_walk_2d_degree_free(1000)

         positions_walker_deg_1 = random_walk_2d_degree_free(1000)

In [102]: fig_2d__deg_rw = plt.figure(figsize=(8,8))

          plt.plot(positions_walker_deg_0["x"], positions_walker_deg_0["y"], label="Walker 0")
          plt.plot(positions_walker_deg_1["x"], positions_walker_deg_1["y"], label="Walker 1")

          plt.title(r"\textbf{Random walk in 2-dimension}", fontsize=22)

          plt.xlabel(r"x position", fontsize=18)
          plt.ylabel(r"y position", fontsize=18)

          plt.legend(fontsize=18, markerscale=10)
```

```
plt.show()
```



6 Let's implement the random walk in a object-oriented method:

6.0.1 First we need to define few classes:

First one is the posistion class which act as our position tracker and have few methods: `move`, `findX`, `findY`, and `distance`

```
In [104]: class position(object):
              def __init__(self, x, y):
                  """x,y are float type"""
```

```
        # assigning the initial position
        self.x = x
        self.y = y

    def move(self,dx,dy):
        """dx,dy are float type: function to make a new position object at the new c
        return position(self.x+dx, self.y+dy)

    def findX(self):
        """Give the x coordinate of the object"""
        return self.x

    def findY(self):
        """Give the y coordinate of the object"""
        return self.y

    def distance(self, other):
        """other is an object from position class: function will calculate their rel
        delta_x = self.x - other.findX()
        delta_y = self.y - other.findY()
        return (delta_x**2+delta_y**2)**0.5


    def __str__(self):
        return "({},{})".format(self.x, self.y)
```

Here I just defined two points (a, b) and I will use the `distance` method:

```
In [105]: a = position(1,2)
          b = position(4,5)

In [106]: b.distance(a)

Out[106]: 4.242640687119285

In [107]: a.distance(b)

Out[107]: 4.242640687119285
```

Great! It seems to work fine!

6.0.2 This is the base class and it's not something useful by itself and it will be inherited.

```
In [108]: # we are going to pass this class to another classes below

          class walker(object):
              def __init__(self, name= None):
                  """assume name is a string"""
                  self.name = name
```

```
        def __str__(self):
            if self.name != None:
                return self.name
            return "Unkown"
```

6.0.3 Here we are going to make two types of walker:

1. Normal walker: which has no preference for any directions.
2. Biased walker: which has some bias toward a particular direction. (in our case in y direction)

```
In [109]: import random

          class Normal_walker(walker):
              def take_step(self):
                  """Taking a random choice out of all the possible moves"""
                  choices_of_steps = [(0,1), (1,0), (0,-1), (-1,0)]
                  return random.choices(choices_of_steps)[0]

          class Biased_walker(walker):
              """Taking a random choice out of all the possible moves"""
              def take_step(self):
                  choices_of_steps = [(0,1.5), (1,0), (0,-0.5), (-1,0)]
                  return random.choices(choices_of_steps)[0]
```

Notice that we have the same name for `take_step` methods under different sub-classes of `walker` which is different when the class is different.

6.0.4 Now we need to define a class for the space that we need to put the walkers in:

```
In [110]: class Space(object):
              def __init__(self):
                  self.walkers={}

              def addWalker(self, walker, pos):
                  """Takes a walker and position class and will add it to our dictionary of wa
                  if walker in self.walkers:
                      raise ValueError("Walker already exist")
                  else:
                      self.walkers[walker]=pos

              def getPos(self, walker):
                  """Will take a walker class and give back the position class assigned to it"
                  if walker not in self.walkers:
                      raise ValueError("No such Walker exist in our space!")
                  return self.walkers[walker]

              def moveWalker(self, walker):
```

```
        """Take a walker class and dependent on what subclass was chosen in defining
        if walker not in self.walkers:
            raise ValueError("No such Walker exist in our space!")
        Delta_x, Delta_y = walker.take_step()
        # moving the walker to new position (class)
        self.walkers[walker] = self.walkers[walker].move(Delta_x, Delta_y)
```

Now that we built up our `position`, `walker`, and `Space` we can make a random walk:

```
In [112]: def walk(space, walker, number_of_steps, log_pos=False):
              """ function for performing a random walk for a given walker
              INPUT:
              -------
                    space is from Space cls
                    walker is from Walker cls
                    number_of_steps is integer>=0

              OUTPUT:
              -------
                    IF log_pos == False:
                                  Function will produce the distance between starting
                                  position of the walker and the last location.

                    IF log_pass == True:
                                  Function will produce a list of all the positions
                                  walker was during the walk.

              """
              # Find the initial postion of the walker in the space
              starting_position = space.getPos(walker)

              # Move the walker in the space
              save_all_pos = []
              for i in range(number_of_steps):
                  pos_=space.getPos(walker)
                  if log_pos:
                      save_all_pos.append((pos_.findX(), pos_.findY()))
                  space.moveWalker(walker)
              if log_pos:
                  return save_all_pos
              return starting_position.distance(space.getPos(walker))
```

In the following we are going to define a function to perform severel random walks:

```
In [115]: def simulate_walks(number_of_steps, number_of_simulations, walker_class_type, origin=
              """
              This is function that runs simulation for given variables:
```

```
        INPUT:
            number_of_steps: How many step the walker should take
            number_of_simulations: How many simulation to run
            walker_class_type: The type of walker class (a subclass of walker)
            origin: Should be an instance of the class position

        Output:
            A list of distances from origins
        """
        our_walker = walker_class_type("walker_1")
        distances=[]
        for i in range(number_of_simulations):
            space = Space()
            space.addWalker(our_walker, origin)
            distances.append(walk(space, our_walker, number_of_steps))
        return distances


    def test_simulation(walk_lenght_array, number_of_simulations, walker_class_type):
        """
        Some sanity checks on the simulations
        """
        for walk_lenght in walk_lenght_array:
            _distances_ = simulate_walks(walk_lenght, number_of_simulations, walker_class
            print(walker_class_type.__name__, " random walk of {} steps".format(walk_leng
            print(" Mean= {}".format(round(sum(_distances_)/len(_distances_),4)))
            print(" Max= {}".format(round(max(_distances_), 4)))
            print(" Min= {}".format(round(min(_distances_),4)))
```

```
In [116]: test_simulation([0,1,2, 10**3, 10**5], 100, Normal_walker)

Normal_walker  random walk of 0 steps  After 100 simulations
 Mean= 0.0
 Max= 0.0
 Min= 0.0
Normal_walker  random walk of 1 steps  After 100 simulations
 Mean= 1.0
 Max= 1.0
 Min= 1.0
Normal_walker  random walk of 2 steps  After 100 simulations
 Mean= 1.3285
 Max= 2.0
 Min= 0.0
Normal_walker  random walk of 1000 steps  After 100 simulations
 Mean= 28.7257
 Max= 91.7061
 Min= 2.8284
```

```
Normal_walker  random walk of 100000 steps  After 100 simulations
 Mean= 286.1797
 Max= 830.3794
 Min= 30.2655
```

```
In [117]: test_simulation([0,1,2, 10**3, 10**5], 100, Biased_walker)
```

```
Biased_walker  random walk of 0 steps  After 100 simulations
 Mean= 0.0
 Max= 0.0
 Min= 0.0
Biased_walker  random walk of 1 steps  After 100 simulations
 Mean= 0.965
 Max= 1.5
 Min= 0.5
Biased_walker  random walk of 2 steps  After 100 simulations
 Mean= 1.3026
 Max= 3.0
 Min= 0.0
Biased_walker  random walk of 1000 steps  After 100 simulations
 Mean= 250.9924
 Max= 303.2375
 Min= 196.4688
Biased_walker  random walk of 100000 steps  After 100 simulations
 Mean= 25029.4813
 Max= 25708.3555
 Min= 24292.8114
```

The next function is going to run the simulation for both walker types, Notice that is general for any number of walker types, here we only defined two but can be extended as well.

```
In [118]: def test_simulate_all_walker_types(walk_lenght_array, number_of_simulations, walker_
              for walker in walker_types:
                  test_simulation(walk_lenght_array, number_of_simulations, walker)

          test_simulate_all_walker_types([0,1,2], 100, [Biased_walker, Normal_walker])
```

```
Biased_walker  random walk of 0 steps  After 100 simulations
 Mean= 0.0
 Max= 0.0
 Min= 0.0
Biased_walker  random walk of 1 steps  After 100 simulations
 Mean= 1.07
 Max= 1.5
 Min= 0.5
Biased_walker  random walk of 2 steps  After 100 simulations
 Mean= 1.2585
```

```
 Max= 3.0
 Min= 0.0
Normal_walker  random walk of 0 steps  After 100 simulations
 Mean= 0.0
 Max= 0.0
 Min= 0.0
Normal_walker  random walk of 1 steps  After 100 simulations
 Mean= 1.0
 Max= 1.0
 Min= 1.0
Normal_walker  random walk of 2 steps  After 100 simulations
 Mean= 1.1681
 Max= 2.0
 Min= 0.0
```

Here we are running the simulation at will find the average of different simulations for the given (fixed) number of steps. We are going to that for the range of steps

```
In [119]: from time import time

          number_of_simulations=100
          number_of_steps_range=300

          ti= time()
          distances= [np.mean(simulate_walks(n,number_of_simulations, Normal_walker)) for n in
          print("Runtime for Normal walker: {} s".format(time()-ti))

          ti= time()
          distances_biased = [np.mean(simulate_walks(n,number_of_simulations, Biased_walker)) f
          print("Runtime for Biased walker: {} s".format(time()-ti))
```

```
Runtime for Normal walker: 9.139477729797363 s
Runtime for Biased walker: 9.050119876861572 s
```

```
In [120]: fig = plt.figure(figsize=(8,8))
          slope=0.25

          plt.plot(range(number_of_steps_range), distances_biased, label="Biased walker (0.5)")
          plt.plot(range(number_of_steps_range), distances, label="Normal walker")
          plt.plot(range(number_of_steps_range), np.sqrt(range(number_of_steps_range)), label="
          plt.plot(range(number_of_steps_range), slope*np.array(range(number_of_steps_range)),

          plt.xlabel(r"Number of Steps", size=16)
          plt.ylabel(r"Average Distance from origin", size=16)
          plt.legend(fontsize=18)

          plt.show()
```