Diffusion
Mathematical Programming with Python

MATH 2604: Advanced Scientific Computing 4
Spring 2025
Monday/Wednesday/Friday, 1:00-1:50pm

https://people.sc.fsu.edu/~jburkardt/classes/python_2025/diffusion/diffusion.pdf

---

Diffusion spreads a disturbance across a medium.

1 Clearing up a problem from last time

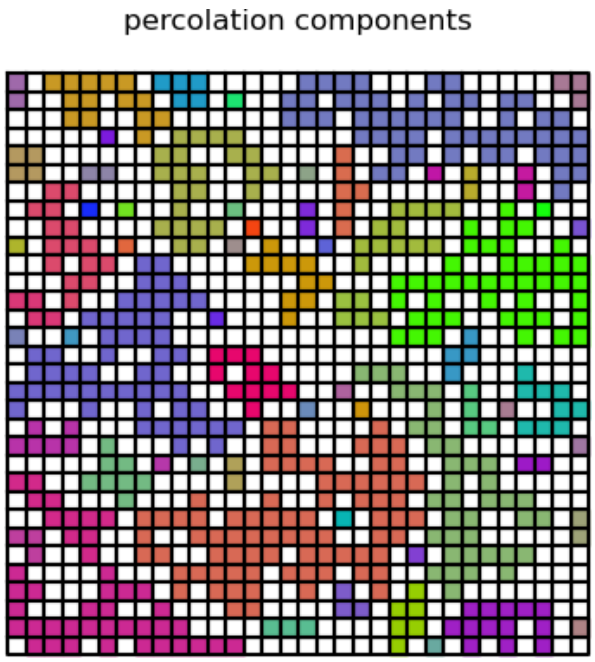
You may recall that I demonstrated a program that examines percolation in a 2D region modeled by a grid. One of the keys to determining whether there was a path for a liquid through the material was to organize the liquid squares into groups or components. All the liquid squares that were neighbors were shown with a single color. However, as we could see in my plots, there were often cases where two squares that were touching had different colors, which clearly indicated a programming mistake.

The algorithm I was using was supposed to be clever and efficient. Sometimes that makes it easy to make a mistake, and hard to find it. After spending a day trying to fix it, I gave up and went back to a simpler idea that we had discussed in class, in which we colored one liquid square, found all its neighbors and gave them the same color, then looked for the next uncolored liquid square, chose a new color for it and all its neighbors.

While this is a little clumsier than the clever algorithm, it turned out to produce plots of the percolation components that were correct. A quick glance at the code is worth a moment, since I used lists, and the `apppend()` and [`pop()` functions in a way that we have not discussed much in class.

```
C = np.zeros ( [ m, n ], dtype = int )
component_index = 0
plist = []

for i in range ( 0, m ):
  for j in range ( 0, n ):
    if ( A[i,j] != 0 and C[i,j] == 0 ):
      plist.append ( i )
      plist.append ( j )
```

```
        component_index = component_index + 1

        while ( plist ):
          j = plist.pop ( )
          i = plist.pop ( )
          C[i,j] = component_index

          if ( 0 <= i-1 and A[i-1,j] != 0 and C[i-1,j] == 0 ):
            plist.append ( i - 1 )
            plist.append ( j )
          if ( i+1 <= m - 1 and A[i+1,j] != 0 and C[i+1,j] == 0 ):
            plist.append ( i + 1 )
            plist.append ( j )
          if ( 0 <= j - 1 and A[i,j-1] != 0 and C[i,j-1] == 0 ):
            plist.append ( i )
            plist.append ( j - 1 )
          if ( j+1 <= n-1 and A[i,j+1] != 0 and C[i,j+1] == 0 ):
            plist.append ( i )
            plist.append ( j + 1 )
```



2 Physics, Mathematics, and Computational Aspects of Diffusion

When we brew tea, we can observe the gradual development of a cloud of tea flavoring the water, spreading from the bag, getting darker until the whole cup has a uniform color. In a similar way, if the bottom of a cooking pan is heated up, gradually the heat spreads over the entire pan. Light a scented candle in your living room, and gradually the smell with spread further and further, becoming uniform, and then gradually fading away. These are all examples of a physical process called diffusion.

Botanist Robert Brown first noticed that tiny pollen particles suspended in water would constantly jiggle, and supposed this occurred because of random collisions of the pollen with individual water molecules. Albert Einstein was one of the researchers who transferred and generalized this idea so that it could be used to describe the behavior of heat, and certain other physical phenomena. You may have seen the somewhat

mysterious heat equation, or diffusion equation, written sometimes as

$$\frac{\partial U}{\partial t} = \alpha \nabla^2 U$$

Instead of this mathematical approach, we will go back to Robert Brown's view, in which diffusion can be represented as many instances of particles on a random walk. Our variable is now $x(t)$, the position of the oarticle at each time. In our simplified model, we will simply assume that a physical system is composed of particles that jump, from one time step to the next, in a random direction and a random stepsize. The random stepsize s will be modeled by a normal random variable s with mean 0 and standard deviation σ. We will also need a unit random direction vector $\overrightarrow{u}$. if $x(t)$ represents the location of some particle at time t, we imagine the value of $x(t + dt)$ to be

$$x(t + dt) = x(t) + s * \overrightarrow{u}$$

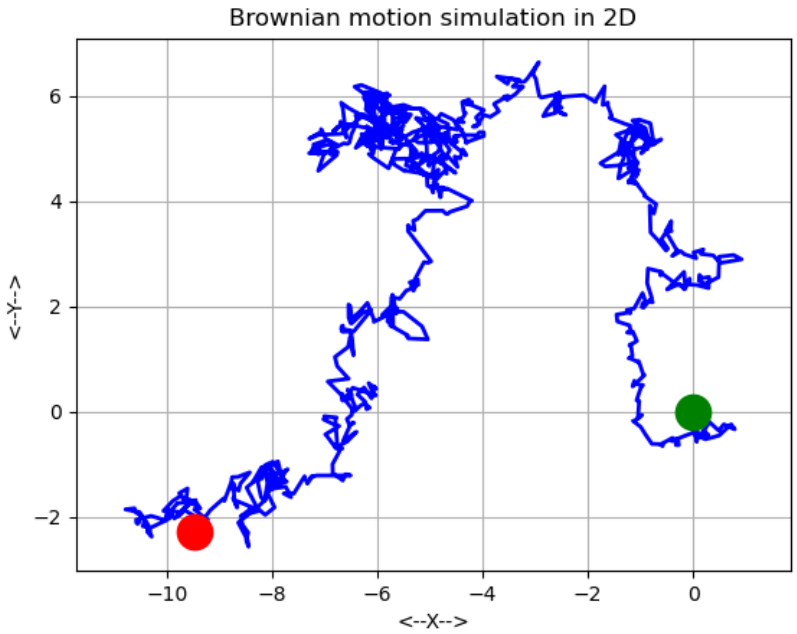
3 Simulating a random walk

Let us suppose we wish to generate a sequence of n points, representing a random walk in m dimensions. Then we will need to specify a starting value x_0, and a characteristic single step size σ. On the i-th step, we compute a step size s and a unit step direction vector $\overrightarrow{u}$. The details of the step direction calculation can be treated specially for $m = 1$ or $m = 2$. For higher dimensions, we must use a little known fact about how to compute a random direction. Here is how, if x is our current position, we can set up the stepsize and direction needed to find the next value:

```
s = sigma * np.random.standard_normal ( )

if ( m == 1 ):
  u = np.random_choice ( [ -1, +1 ] )
elif ( m == 2 ):
  t = 2.0 * np.pi * np.random.random ( )
  u = [ np.cos ( t ), np.sin ( t ) ]
else:
  u = np.random.standard_normal ( m )
  u = direct / np.linalg.norm ( u )

x[i] = x[i-1] + s * u
```

This assumes we want to keep the individual points in the walk, so that we can create plots, or perform other kinds of analysis. If not, we can simply use a single value x which is immediately overwritten on every step.

For example, here is a sample random walk of 1001 steps in a 2D region:



Particle started at green dot, and was halted at the red dot.

We will generate these walks in dimensions 1, 2 or 3. While an individual random walk seems highly irregular, we will see, by computing and averaging many examples, that there are some underlying predictable patterns to be unconvered.

4 A random robot walker

Bellefield Robotics has designed a food truck robot named "BumbleBee" which offers food for sale at the corners of street blocks along Fifth Avenue. Its home station is at Bellefield Avenue. BumbleBee simply wanders up and down Fifth Avenue, from block to block, looking for customers. As a safety feature, it shuts down if it reaches a distance of 10 blocks from home base.

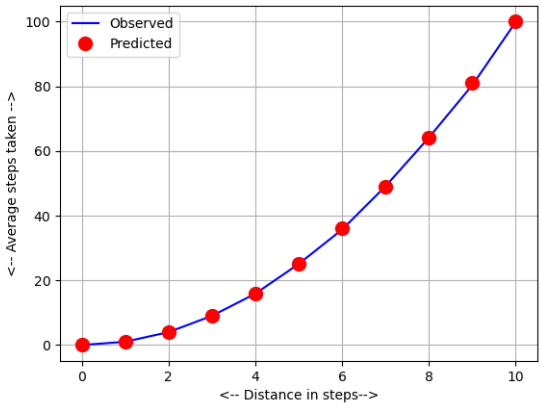
On Friday morning, BumbleBee starts at Bellefield Avenue, and chooses a random direction, east or west, to go to the next block. Today there are no customers, so after waiting at one block for five minutes, Bumblebee again randomly moves east or west to another block.

We may reasonably assume that at some point in the day, BumbleBee will randomly reach a distance of 10 blocks, and shut down. On average, when will this happen? Clearly, it takes a minimum of 10 * 5 minutes to get that far from home base, and obviously, there is theoretically no maximum amount of time that could be involved. But realistically, would we have to wait two hours, two days, or what?

We can easily simulate one instance of this walk:

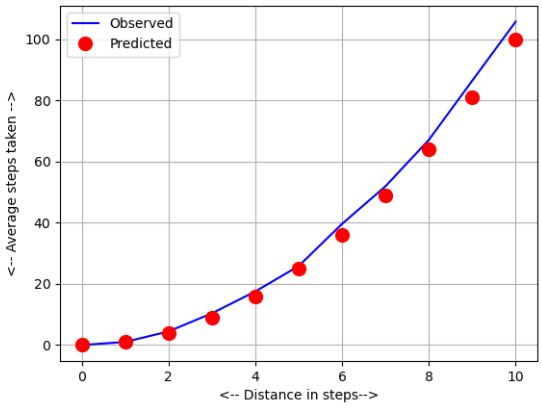
```
def walk_1d ( ):

  directions = [ -1, +1 ]
  location = 0
  blocks = 0

  while ( -10 < location and location < 10 ):
    blocks = blocks + 1
    location = location + np.random.choice ( directions )

  return blocks
```

If we simulate BumbleBee's adventure five times, we see that walks of length 220, 42, 32, 118 or 48 blocks, to reach a destination that is only 10 blocks away. This is a little irregular and mystifying. The first thing we need to do is to generate lots of samples of this walk and find the average time it takes to reach a distance of 1 block, 2 blocks, ..., 10 blocks.



Surprisingly, the average data is very regular, and suggest that to reach d blocks in a random fashion, we actually typically have to walk d^2 blocks. Thus, on average, BumbleBee will shut down after 5×100 minutes, or a bit more than 8 hours after starting.

Now let's suppose that BumbleBee is moved to a new location, perhaps Manhattan, and let us suppose this is an area with a perfectly regular square street grid. Now, at each step, Bumblebee is allowed to move one block north, south, east or west. Again, we draw a limiting circle of radius L blocks from the origin, and decree that Bumblebee must shut down upon hitting this boundary. If we run this experiment many times, what do we see?



Surprisingly, the fact that it takes about L^2 steps to reach the boundary holds for both the 1D and 2D cases. You should not be surprised to hear that the 3D case works in the same way.

Perhaps more surprisingly, this fact suggests that if it takes 3 minutes to cook a steak that is one inch thick, it will take 4 times as long (12 minutes) to cook a steak that is twice as thick. That is because heat is a kind of diffusion or Brownian motion which makes progress at a rate proportional to the square root of time.

4.1 3D Plotting Issues

If we are interested in 3D plots, the ttmatplotlib library needs an auxilliary library called `Axes3D`, and the rules for construcing a 3D plot are distinctly different. Here, for example, is how we would make the plot of a 3D random walk:

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure ( )
ax = fig.add_subplot ( 111, projection = '3d' )

ax.plot    ( x[:,0],   x[:,1],   x[:,2],   'b', linewidth = 2 )
ax.scatter ( x[0,0],   x[0,1],   x[0,2],   c = 'g', marker = 'o', s = 100 )
ax.scatter ( x[n-1,0], x[n-1,1], x[n-1,2], c = 'r', marker = 'o', s = 100 )
ax.grid ( True )
ax.set_xlabel ( '<--X-->' )
ax.set_ylabel ( '<--Y-->' )
ax.set_zlabel ( '<--Z-->' )
plt.title ( 'Brownian motion simulation in 3D' )
plt.show ( )
```

You can see that most of the familiar commands have a new form, while still involving the same input.

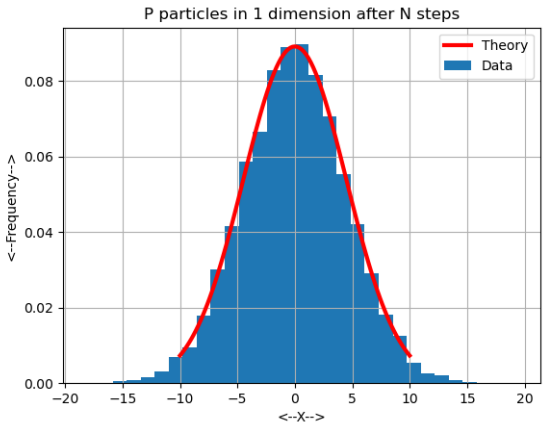
5 The distribution of random walkers after N steps

For complex processes like heat transfer, we have to imagine many random walkers, all released at the same time, and wandering away in all directions. We can actually get an idea of how the walkers scatter using graphics.

For a 1D random walk, it makes sense to draw a histogram of the location of all the walkers.

```
plt.hist ( x, bins = 31, density = True )
plt.grid ( True )
plt.xlabel ( '<--X-->' )
plt.ylabel ( '<--Frequency-->' )
plt.title ( 'P particles in 1 dimension after N steps' )
```

which results in

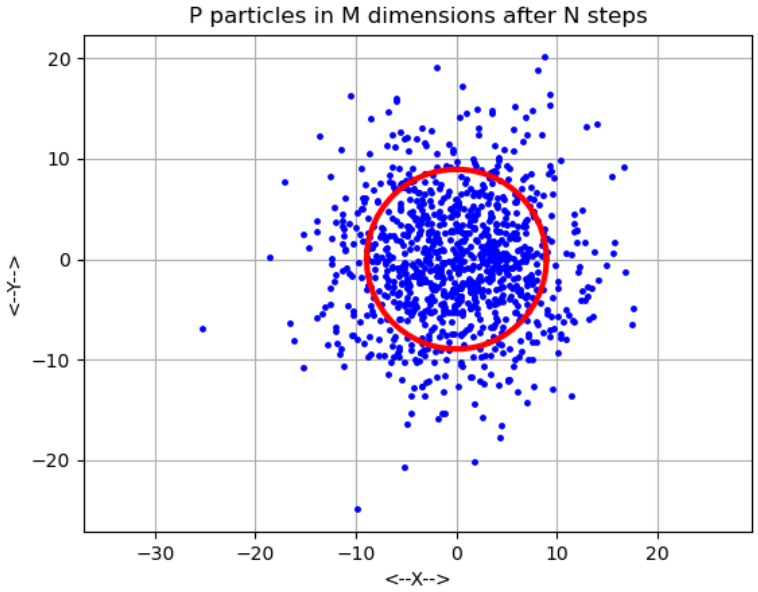


where we have included a fit to the normal distribution curve.

For a 2D random walk, we can show the individual locations

```
plt.plot ( x[:,0], x[:,1], 'b.', markersize = 5 )
plt.grid ( True )
plt.xlabel ( '<--X-->' )
plt.ylabel ( '<--Y-->' )
plt.title ( 'P particles in 2 dimensions after N steps' )
plt.axis ( 'equal' )
```

which results in



where we have included a circle with radius $\sqrt{2}\sigma$, which captures most of the data qne suggests it is organized as a normal distribution.

A similar image can be made of a 3D random walk, which shows how the data forms a rough sphere. The interactive image can be rotated to make a more convincing demonstration than is possible in print.