HY514: Problem Solving Environments

Assignment 3: Brownian Motion*

Elias Houstis

Due Date:

Introduction

The first part of the third assignment was adopted from 600.112: Introduction to Programming for Scientists and Engineers explores the phenomenon of brownian motion and other so-called random walks more generally.

There are three things to do: First you'll write a program that primarily simulates brownian motion of particles in a liquid, also keeping track of the shortest and longest distance that the particles travel. Second you'll extend that program to a more complex simulation that approximates (very roughly!) the process by which cancer cells spread into the blood stream. Third you'll explore the random walk of a particle within a boundary, modelling reflection when collisions occur.

There are detailed submission instructions on Piazza which you should follow to the letter! You can lose points if you create more work than necessary for the graders by not following the instructions.

Background

The phenomenon of brownian motion was first described in 1828 by Robert Brown, a biologist. Brown observed that pollen suspended in water moved around seemingly at random without any obvious cause. A convincing explanation for this observation was finally given in 1905 by Albert Einstein. Einstein theorized that the pollen are constantly bombarded by water molecules, and that slight variations in how many molecules hit from various directions would (overall) lead to the motion Brown had observed.[1] Jean Perrin eventually verified Einstein's predictions experimentally and in 1926 won a Noble Prize for this work.

1 Just Brownian Motion (20% = 10 points)

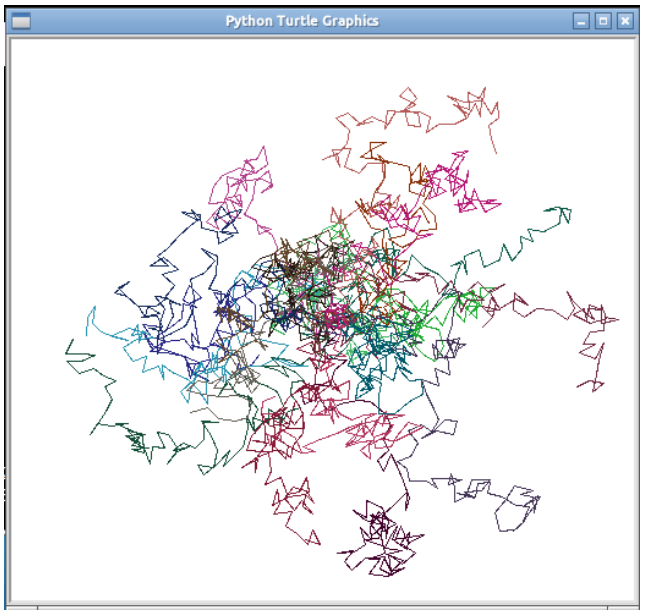
The first program you will write simply simulates Brownian motion of particles in a liquid. We will visualize the simulation using Python's turtle graphics module once again. Please call your program `motion.py` and nothing else! Figure 1 shows what the output of your program will look like, at least approximately: since we are simulating a random process, the paths taken by each particle will be different every time the simulation is run.

For this and the following programs, you will need to use pretty much all of the Python concepts covered so far in the course. As always, however, we will lead you through the problems rather slowly and with a lot of advice on how to proceed, so you should be okay as long as you follow along diligently. And as you're probably used to by now,

* Disclaimer: This is not a course in physics or biology or epidemiology or even mathematics. Our exposition of the science behind the projects cuts corners whenever we can do so without lying outright. We are not trying to teach you anything but computer science!

1. While the actual physics underlying brownian motion is quite fascinating, we'll avoid getting into too detailed a discussion here. In addition, Brownian motion takes place in gasses as well as liquids. There will probably be another assignment that will focus on the physics of liquids and gases exclusively.

Figure 1 Output of the `motion.py` program for 20 particles, each drawn in a different color.



we'll start with a very basic first version of the program:

```
"""Simulation of brownian motion."""

import turtle

def main():
    turtle.setup()
    turtle.done()

main()
```

There is one noteworthy addition here that we haven't seen before: The first line contains a short description of the program we are about to write between triple-double-quotes.[2] This is Python's way of putting documentation into a program. From now on, each and every program you write for the course should contain a brief description of what the program does in this format.

Of course the first version of the program doesn't do anything, so we'll now have to decide how the simulation should work. Let's first worry about an individual particle and its motion. If we look at the particle again and again at constant intervals, it will have travelled random distances (each within some reasonable range) from its previous position every time. Furthermore, it will have travelled that distance in a random direction! Translated into Python's turtle graphics, this means that we can move the particle using `turtle.forward` by some random integer amount in a fixed range, say between 10 and 15 units. Also, between each move, we have to randomly determine a new orientation for the turtle.

Luckily Python has a module called `random` that can help us out here. Carefully study the following Python Shell interaction:

```
>>> import random
>>> random.random()
0.686507793058002
>>> random.random()
0.3543865018260042
>>> random.random()
0.7794135114537576
>>> random.randint(50, 100)
79
>>> random.randint(50, 100)
60
>>> random.randint(50, 100)
100
```

Every time we call the `random.random` function, it will return a different floating-point number between 0 and 1, each picked uniformly at random. Every time we call the `random.randint(a,b)` function, it will return a random integer between `a` and b, inclusive, with all possibilities being uniformly distributed.[3] There are several ways to generate random values using the random module. For example, if we want to generate a random real number between 0 and 9, we can simply multiply the next random float by 10:

```
>>> random.random() * 10
5.941385596081145
>>> random.random() * 10
4.0655598261150185
```

In choosing each step length for the turtle, we will generate a random integer between 5 and 15. Giving the turtle a random orientation simply requires that we call `turtle.left` with a random integer between 0 and 359 representing the angle to turn. Now that we know how to simulate a single move of the particle, and we can simply perform a

2. You can also use triple-single-quotes instead: `'''` will work just as well as `"""` does. Regardless which notation you prefer, it's important that you use one or the other consistently.

3. At least "as random as possible" we should add. The task of generating random numbers with a deterministic machine such as your computer is not a simple one. But for our purposes the numbers are certainly "random enough" as it were.

given number of steps to simulate a longer period of time:

```
"""Simulation of brownian motion."""

import turtle
import random

def particle(steps):
    """Simulate one particle for
    the given number of steps."""
    for _ in range(steps):
        angle = int(random.random() * 360)
        turtle.left(angle)
        turtle.forward(random.randint(10, 15))

def main():
    turtle.setup()
    particle(100)
    turtle.done()

main()
```

Note that we also added a sentence of documentation to the function we wrote; from now on, each and every function you write for the course (other than main) should contain a good description of what the function does.

Now that we know how to simulate the motion of one particle, we want to calculate the total distance that the particle travels as well. This is an example of the accumulator pattern that is very common in computing. It's similar to the homework point sum example that we explored earlier in the course. We will need a variable to sum the distances of all the steps within the particle function and **return** this value. Naturally, the sum should be initialized to 0. Within the **for** loop, we need to add the distance of each step to this sum, which means we will need to store each random value in a variable before moving forward. Here is a new version of the program which also prints the distance a particle has travelled in main. Note that sum is the name of a common function in python, so we use **total** as our variable name instead.

```
"""Simulation of brownian motion."""

import turtle
import random

def particle(steps):
    """ Simulate one particle for
        the given number of steps.
        Calculate and return the
        total distance travelled.
    """
    total = 0
    for _ in range(steps):
        angle = int(random.random() * 360)
        distance = random.randint(10, 15)
        turtle.left(angle)
        turtle.forward(distance)
        total = total + distance
        # we can write instead: total += distance
    return total   # this becomes the value of the function call

def main():
    turtle.setup()
    print particle(100)
    turtle.done()

main()
```

We now have the core of the simulation program we set out to write. Notice that we updated the documentation for the method to include the total distance calculation. (As an experiment, try indenting the return statement another 4 spaces. What happens? Why?) All we have to add now is the ability to simulate multiple particles, to pick random colors for each of them, and to compare the distances they travel to determine the longest and the shortest paths.

Once again we snuck something new into the code above: the lines beginning with the # character are Python's way of including comments in a program. The documentation we added earlier is intended for anyone who wants to use the program (or a function from it), so think of it as "advertising copy" for users or customers. Comments, on the other hand, are notes from one programmer working on a piece of code to another programmer working on the same piece of code.[4] Comments are frequently used when the purpose of a snippet of code may not be entirely obvious at first sight.

While it was easy to say "always include documentation for programs and functions from now on" there is no such rule for comments. Ideally everything you do is so simple and obvious that you don't need comments. But every now and then it's a good idea, especially with a piece of code that

4. That mysterious "other" programmer may in fact be you a few weeks later when you cannot remember anymore why you did a certain thing in a certain way!

uses "exotic" functions for a non-obvious purpose.

When running this version of the simulation, you probably noticed that even just one particle takes a very long time to draw; you probably also noticed the poor turtle spinning around again and again like it was drunk. If we ever want to simulate a bunch of particles, we need to somehow speed up the process of drawing. Here is what we do:

```
"""Simulation of brownian motion."""

import turtle
import random

...

def main():
    turtle.setup()

    # the following three calls speed
    # up drawing significantly
    turtle.hideturtle()
    turtle.speed(0)
    turtle.tracer(0)

    print particle(100)

    turtle.done()

main()
```

In brief, `turtle.hideturtle` disables the little triangle that represents the turtle: by not having to draw it (and all its drunken turns) things go much faster already. The `turtle.speed` function sets the drawing speed; by default Python's turtle graphics go slow so that you can follow what is going on as the program runs; calling the function with a speed of 0 essentially means "no more delays please." Finally, `turtle.tracer` with a parameter of 0 means that we really only draw the finished picture, not all the intermediate states. When you run the program with these calls included, the final image should simply appear out of nowhere: it's that fast.

Next let's figure out how to pick a random color. For this you need to understand that all the colors you see on your screen are "mixed together" from just three basic colors: red, green, and blue (RGB values). Yellow, for example, results when you combine red and green, whereas purple results when you combine red and blue. The `turtle.color` function we've met before can be called with three floating-point numbers between 0 and 1, specifying the "percentage" each of red, green, and blue to "mix" into a new color. We already know how to use `random.random` to generate a random number between 0 and 1, but we want to avoid colors that are "too light" or "too close to white" since the background of our window is white.[5] It's easy to fix this: Just multiply the random number by 0.75 and none of the components will be "too light" anymore. Here is the code:

```
"""Simulation of brownian motion."""

import turtle
import random

def randomColor():
    """Pick a random color
    that is dark enough."""
    red = random.random() * 0.75
    green = random.random() * 0.75
    blue = random.random() * 0.75
    turtle.color(red, green, blue)

# particle method goes here

def main():
    turtle.setup()

    # speed up statements go here

    randomcolor()
    print particle(100)

    turtle.done()

main()
```

Each and every time you run the program now, the particle should get drawn in a different color.

The next thing we have to add is code to simulate a number of particles instead of just one particle. We'll simply do this one particle at a time for now, later we'll see how we can do it—in a more realistic way—for several particles at once. So we want to simulate n particles in a row, each of which will be drawn in a different color. For simplicity we'll assume that each particle starts in the center of the screen, and the `turtle.home` function does exactly that: move the turtle back from where it currently is to the center. We just have to make sure that we do a `turtle.up` before going home and a `turtle.down` after to avoid drawing a line that we don't want. Here is the code:

5. White results when all of red, green, and blue are at 100% intensity; black results when all of red, green, and blue are at 0% intensity.

```
"""Simulation of brownian motion."""

import turtle
import random

# randomColor definition

# particle definition

def simulate(particles, steps):
    """ Simulate the given number of
        particles, each for the given
        number of steps; each particle
        starts in the center with a
        new random color.
    """
    for _ in range(particles):
        turtle.up()
        turtle.home()
        turtle.down()
        randomColor()
        print particle(steps)

def main():
    """Run an entire simulation."""
    turtle.setup()
    ...
    simulate(20, 100)
    turtle.done()

main()
```

Now the last step is to do something with all those distances that are being printed as we simulate multiple particles. Our goal is to find the shortest and the longest distances. We'll do this in the simulate function as well. First we need to initialize variables to keep track of the shortest so far and the longest so far. Then we can compare the distance for each particle to those values using decision statements, and update them if necessary.

But how do we initialize those values? We need to make the longest variable as small as possible, so we can use 0. However, we need to make the shortest as big as possible, to make sure that the first simulated distance will update it. Since we know that each step is at most length 15, we can multiply that by the number of steps and add a little extra bit to be sure. We will then print the shortest and longest values in the simulate function. (Later in the course we'll learn how to return multiple values from a function instead.) Here is the revised simulate function. If you want to check that shortest and longest are being calculated correctly, then include a statement to print each distance as we had before. Once you are sure all is well, you can comment that line out by putting a # at the start.

```
def simulate(particles, steps):
    """ Simulate the given number of
        particles, each for the given
        number of steps; each particle
        starts in the center with a
        new random color. Calculate
    and
        print the shortest and
        longest paths travelled.
    """
    shortest = 15 * steps + 5
    longest = 0
    for _ in range(particles):
        turtle.up()
        turtle.home()
        turtle.down()
        randomColor()
        dist = particle(steps)
        # print dist  # keep this in
            for testing
        if dist < shortest:
            shortest = dist
        if dist > longest:
            longest = dist
    print 'shortest distance:',
        shortest
    print 'longest distance:', longest
```

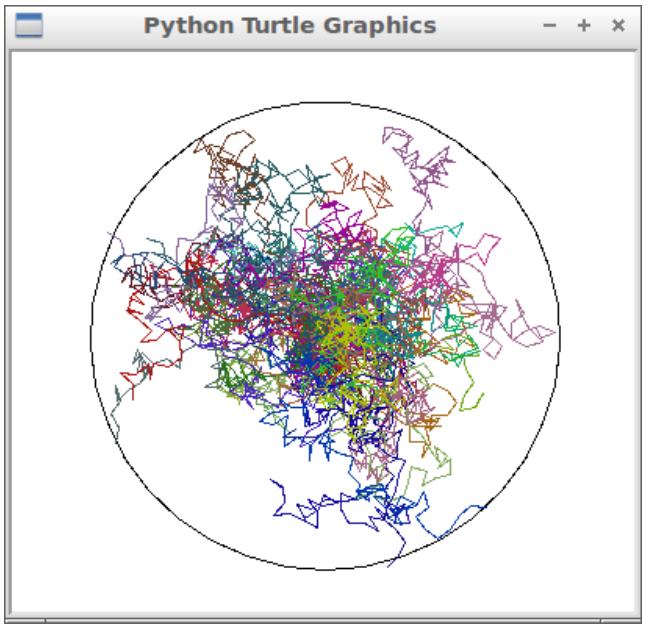
One more thought question before we leave this version of the problem: what value would you expect to get if we calculated the average distance a particle travels? Your answer should take into consideration the fact that random values by default are uniformly distributed in their specified range. Just for fun, add a little code to calculate and print the average. How close was your guess to the simulated results?

2 Escaping Cells (30% = 15 points)

The second program you will write extends the one from above to simulate (very roughly!) the process by which cancer cells spread into the blood stream.[6] Please call your program **escape.py** and nothing else! Figure 2 shows what the output of your program will look like, at least approximately; aside from this visualization, you'll also

6. We already had a disclaimer about the accuracy of the scientific background for these assignments. Here we should add that this is most definitely not an accurate model of metastasis.

Figure 2 Output of the `escape.py` program for 40 cancer cells in an organ of radius 150.



print out how many of the simulated cancer cells actually "escaped" into the blood stream.

Roughly speaking, cancer develops when a tissue cell is genetically damaged to produce a cancer stem cell. Cancer stem cells reproduce, for example inside an organ, and can eventually spread into the blood stream. Our very simplified model of this process assumes that cancer cells originate in the center of a perfectly round organ. We further assume that cancer cells move randomly inside the organ, and that those who make it to the boundary of the organ can enter the blood stream. The question is how many cancer cells will "escape" the organ and enter the blood stream in a given period of time.

Your new simulation can in large parts be based on the solution for the previous problem. You'll simply have to carefully modify and extend the code you already have. Here are some of the things you must have in your final program for the cancer simulation:

- All the identifiers/names in the program should be made to fit the new simulation, for example you want to talk about cells intead of particles. All the documentation strings should be adjusted to reflect the modified functions and their new meanings. (The program as a whole should be coherent and consistent in itself.)
- A function `circle` that takes the radius for a circle and draws it around the center of the window. You will probably have to play with the `turtle.circle` function for a while before you figure out how to write your `circle` function correctly.
- The `cell` function will need a new parameter for the radius of the organ boundary; if a par- ticle ever leaves the organ, the function should return immediately without finishing the re- maining steps. The `turtle.distance` function will be helpful for this, play with it for a while to figure out how to use it.
- The `simulate` function will also need the radius of the organ boundary since it has to pass that information on to the `cell` function. Furthermore it has to draw the boundary itself using the `circle` function you wrote. Finally, the simulate function should return the number of cancer cells that escaped from the organ. Remember that you can use `turtle.distance` to see if the simulation of one cell ended up outside the organ. You will need to check this after each step that a cell takes. Simply count the number of cell simulations that left the organ and return the total.

The new `main` function should look pretty much like this:

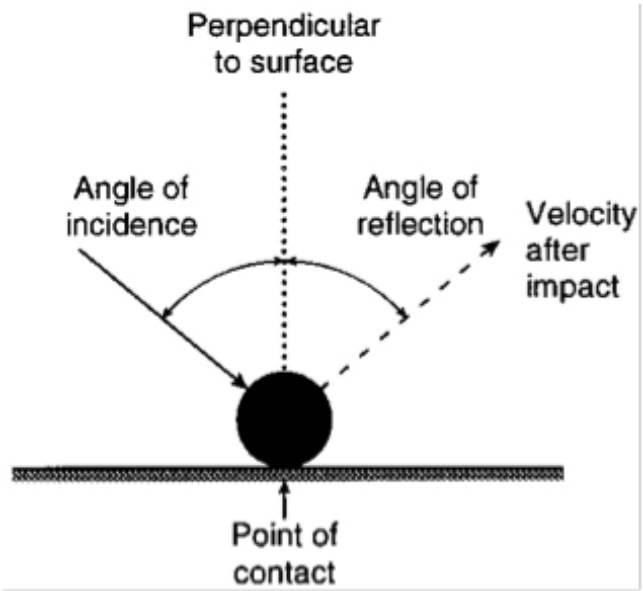
```
def main():
    """Run an entire simulation."""
    turtle.setup()

    # the following three calls speed
    # up drawing significantly
    turtle.hideturtle()
    turtle.speed(0)
    turtle.tracer(0)

    cells = 40
    escaped = simulate(cells, 100, ←
        150)
    print escaped, "out of", cells, "←
        cells escaped"

    turtle.done()
```

Figure 3 Angles of incidence and reflection.



3 Random Walks with Collisions (50% = 25 points)

For this final part of the assignment you'll write a program to simulate a particle in a box, using reflection to model collisions against the boundaries of the box. [7] Please call your program `collisions.py` and nothing else!

Now, we will add a little more physics! Ideally, when a suspended particle hits an immovable object such as the container it is in, the particle will conserve its kinetic energy and rebound off appropriately. This interaction is referred to as an elastic collision. Regarding the collisions, there are two things you need to know for this assignment:

- The angle of incidence is equals to the angle of reflection (see Figure 3).
- Treat each randomized step as a vector (this will also help you in your path-planning hint: angles). This means that after you have randomized an angle and distance that the particle will be traveling, the particle must finish traveling that entire distance, even if it collides with the boundary mid-way.

Specifically, you should create several functions to do most of the processing: boundary, particle, checkCollision, and collisionMove. (You may want to reuse some of your code from assignment 2 and the parts above.)

The `boundary` function is very straightforward: draw a red square box of a given size, centered in the graphics window. In your final program version, call this function to draw a box wtih side length of 400.

Next you will need a `particle` function to draw and simulate the motion of a particle within a square boundary. This function should have parameters that allow you to specify the size of the box, the number of steps that the particle takes, and the lower and upper limits for the random step lengths of the particle (eg, 10 and 15 in part 1 of this assignment). This function will treat each randomized step (angle and step length) as a vector. This means that after you have randomized an angle and distance (step length) that the particle will be traveling, the particle must finish traveling that distance, even if it collides with the boundary. For example, if the particle is 20 units below a wall boundary and you generate the random vector to travel directly upwards for 50 units, it will travel 20 units upward until the particle hits the wall, then finish the vector by traveling 30 units away (down) from the wall.

After generating the random vector (angle and step length), you'll need a function (`checkCollision`) that can check if your generated vector will collide with a boundary, and then move accordingly. If no collision is imminent, it can move the full distance for this step. If a collision will occur, then you'll need to call the new `collisionMove` function described below instead.

The `checkCollision` function will need two parameters: the distance to travel and the size of the boundary. It should return an integer that indicates with which wall (if any) the particle will collide. Here is the exact documentation for it:

```
def checkCollision(dist, size):
    """
    Check if the particle is
    going to go out of bounds.
    Returns a number based on
    which bound is getting hit:
    0 - No collision
    1 - Top
    2 - Right
    3 - Bottom
```

7. Disclaimer: It is important to realize that even though we will be tracking the motion of a single particle, this random Brownian motion would not be possible without the help (collisions) of the various other particles in the same medium.

```
    4 - Left
    """
```

To calculate where the turtle's generated vector would take it without boundaries, start with vector addition. For example, newX = currentX + cos(vectorAngle)∗ vectorDistance. For the turtle
module, look-up how to get the turtle's current direction (angle), and location coordinates. Note: It is not enough to simply check the endpoint in this case, you want to figure out some way to ensure that the first wall the turtle comes in contact with is the wall returned by the collision function. So pretend to move the turtle one pixel at a time along its vector and check each new location for a wall collision.

The **collisionMove** function will be used to move the turtle when a collision is imminent. It needs to calculate the distance that the turtle has unti it hits the indicated boundary, moving that far so that the turtle is on the boundary. Afterwards, you want to turn the turtle according to the angle of incidence so that it reflects appropriately (knowledge of what boundary was hit can help with this), randomly change the color of the turtle to indicate a wall has been hit, and finish traveling the rest of the initial vector magnitude. But wait! There can be multiple collisions for a randomly generated vector (especially if it's length is larger than the box size). How can you account for this? We'll use a special trick called recursion which is when a function calls itself.

Here are some pieces of the **collisionMove** function to get you started, including the recursive method call at the end. You'll have to fill in the missing pieces yourself, indicated by # more here.

```
def collisionMove(dist, size):
    """
    Special move for when it appears
    that the particle will collide
    with a boundary, where dist is how
    far the particle should travel
    and size is the box side length.
    """
    s = size / 2
    bound = checkCollision(dist, size)

    # more here

    if bound == 1:
        untilwall = (s - turtle.ycor()
            ) / math.sin(math.radians(
            angle))

    # more here

    elif bound == 4:
        untilwall = (-s - turtle.xcor
            ()) / math.cos(math.radians
            (angle))

    # more here

    dist = dist - untilwall
    if checkCollision(dist, size) ==
        0:
        turtle.forward(dist)
    else:
        collisionMove(dist, size)
```

Figure 4 Output of the **collisions.py** program with one really long step.

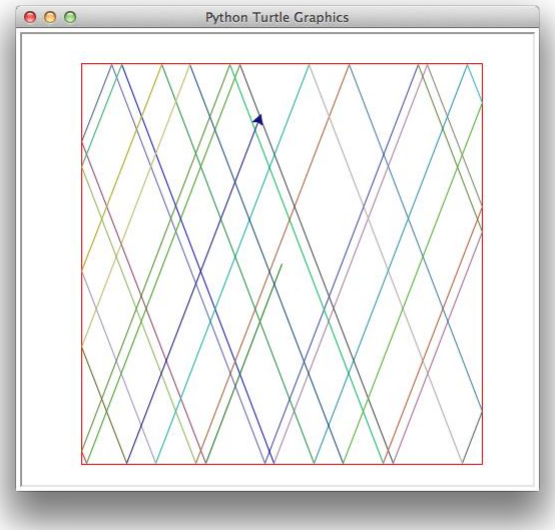
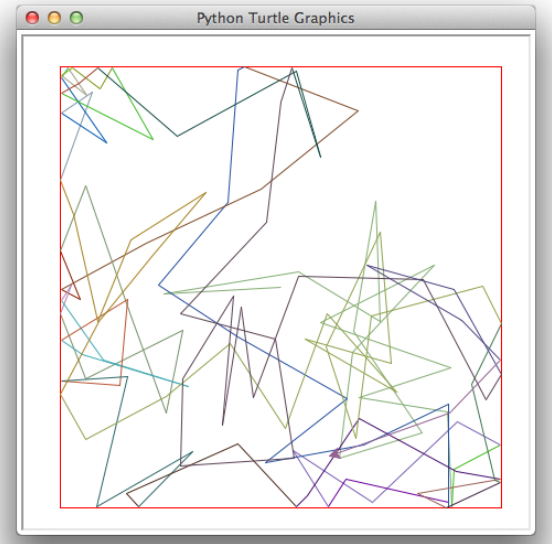


Figure 4 shows what your program will look like if you make your step length really large and not random (particle(400, 1, 10000, 10000)). In this case, the particle is tracing one long path and keeps bouncing off various walls as it moves.

Figure 5 shows what your program will look like if you make your step size smaller than the box and random (particle(400, 100, 75, 125)). In this case, the particle's color changes each time it bounces off the wall and each time it starts a new step, changing direction randomly then as well.

Set up your main program so that it draws both. Depending on how exactly you do this, you may need to close the graphics window after the first particle finishes drawing in order to enable the next drawing to start. Or you may have them both draw in the same window.

Figure 5 Output of the `collisions.py` program with multiple smallish random steps.



Here are some thought questions for you. What will happen if you make one really long step and the initial angle happens to be 0, 90, 180 or 270? In general, what angles and step-length combinations might cause the same lines to be retraced with one long path?

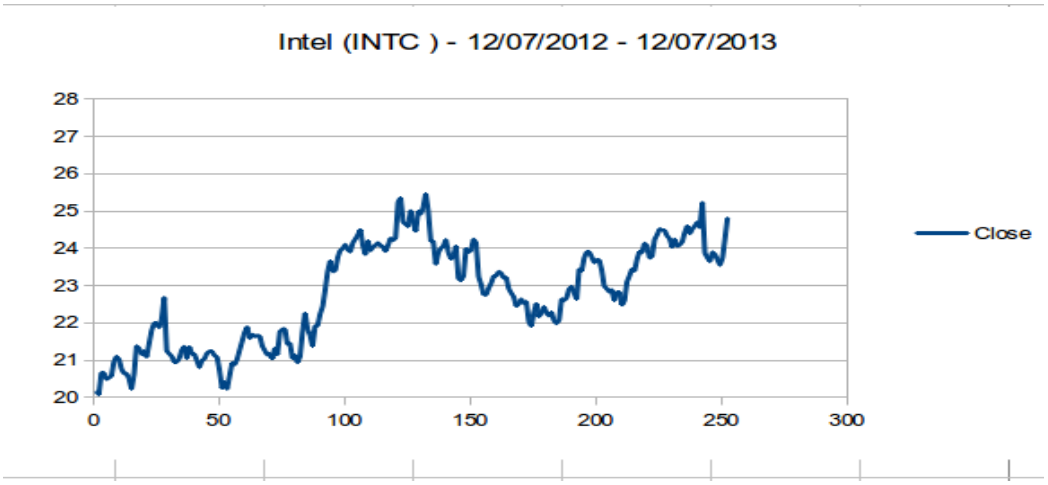
Reference: http://www.cs.jhu.edu/~joanne/cs112/

Python in Finance (50 points)

Modeling Asset Prices with Geometric Brownian Motion

In this note, show how classes from Monte Carlo framework can be utilized to model the path an asset's price can take over a period of time, such as that depicted in the one-year Intel (INTC) stock chart below.

Why would we want a model of asset prices? There are a number of good reasons. One is that a model of asset price dynamics is essential to the valuation of derivatives, such as equity and index options. Secondly, such a model is a powerful tool for risk management. Simulated asset prices can be used to create a range of what-if scenarios with which to calculate a portfolio's aggregate market risk exposure as measured by metrics such as Value at Risk (VaR). And thirdly, simulated prices that conform to historical asset return parameters (e.g. annualized mean and standard deviation) can be employed as market data for back-testing trading strategies.



So now that we appreciate the why, let's look at the how. To start, let's briefly address the theoretical foundations of asset price dynamics- the stochastic process. Informally, a stochastic process is a function of one or more time varying parameters where at least one of the parameters is non-deterministic; its values correspond to a sequence of independent random variables drawn from a selected probability distribution. More precisely, the particular stochastic process that describes the evolution of stock prices is termed Geometric Brownian Motion (GBM).

Geometric Brownian Motion can be formulated as a Stochastic Differential Equation (SDE) of the form:

$$dS_t = rS_t dt + \sigma S_t dZ_t$$

where S is the stock price at time t, r represents the constant drift or trend (i.e. annual return) of the process and σ (sigma) represents the amount of random variation around the trend (i.e. annualized standard deviation of log returns). Intuitively, r can be viewed as the 'signal', while sigma is the 'noise' of the GBM stochastic process. What this equation tells us is that the change in a stock's price over a small discrete time increment (dt) is a function of the stock's return (r) and the stock's volatility (sigma), where the volatility is scaled by the output of a Wiener process (dZt). The Wiener process essentially provides random numbers in accordance with a given (usually Gaussian) probability distribution. For more information on Geometric Brownian Motion http://en.wikipedia.org/wiki/Geometric_Brownian_motion.

An exact discretization scheme of the stochastic differential equation is given for t>0 by

$$S_t = S_{t-\Delta t} \exp((r - 0.5\sigma^2)\Delta t + \sigma\sqrt{\Delta t}\, z)$$

with z being a standard normally distributed random variable.

Following find an implementation of the process with pure python. You are supposed redo it using NumPy which provides a much faster implementation through vectorization/matrix notation, NumPy code is much more compact, easier to write, to read and to maintain performance: NumPy is mainly implemented in C/Fortran such that operations on the NumPy level are generally much faster than pure Python. Time the two versions and compare them.

Simulation with Pure Python

First, we import needed functions and define some global variables.

```
#
# Simulating Geometric Brownian Motion with Python
#
from time import time
from math import exp, sqrt, log
from random import gauss

# Parameters
S0 = 100; r = 0.05; sigma = 0.2
T = 1.0; M = 50; dt = T / M
```

Then we define a function which returns us I simulated index level paths.

```
# Simulating I paths with M time steps
def genS_py(I):
    ''' I: number of paths '''
    S = []
    for i in range(I):
        path = []
        for t in range(M + 1):
            if t == 0:
                path.append(S0)
            else:
                z = gauss(0.0, 1.0)
                St = path[t - 1] * exp((r - 0.5 * sigma ** 2) * dt
                                       + sigma * sqrt(dt) * z)
                path.append(St)
        S.append(path)
    return S
```

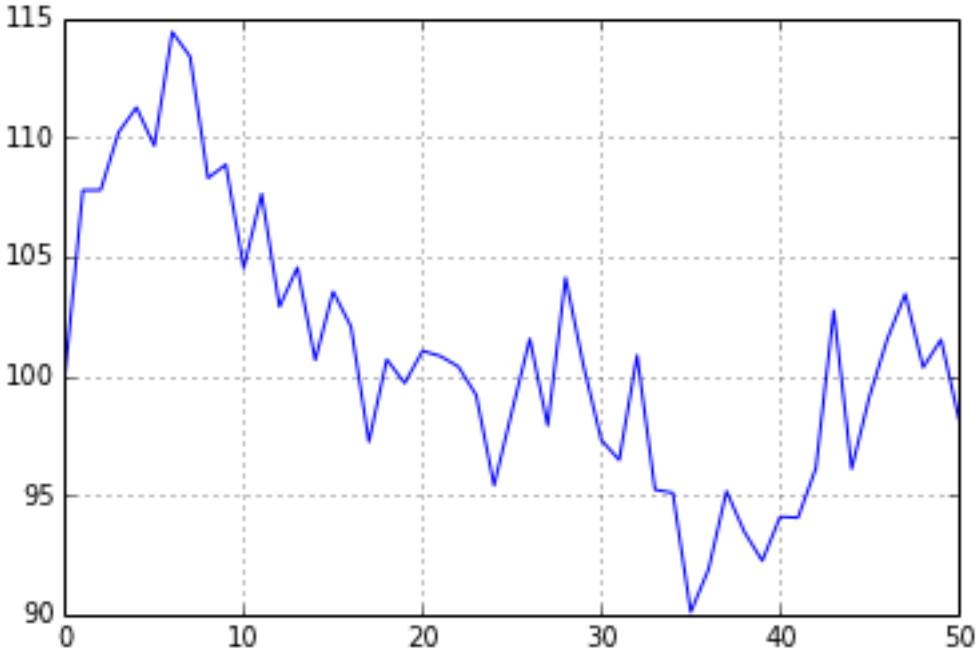
Let's see how long the simulation takes.

```
I = 100000
%time S = genS_py(I)
```

Wall time: 12.2 s

Plot the results of simulation

```
plot(S[10])
grid(True)
```



Reference:
http://github.com/jrjohansson/scientific-python-lectures