math407::cp

Problem 1

Find a procedure for sampling uniformly on the surface of the sphere.
(a) Use computer to generate a thousand points that are random, independent, and uniform on the unit sphere, and print the resulting picture.
(b) By putting sufficiently many independent uniform points on the surface of the Earth (not literally but using a computer model, of course), estimate the areas of Antarctica and Africa, compare your results with the actual values, and make a few comments (e.g. are the relative errors similar? would you expect them to be similar? if not, which one should be bigger? how does accuracy improve if you use more points? etc.)

For the problem a), I use `python` to generate a sphere and use a uniform number generator to generate a **theta** value from 0 - 2π and use another uniform number generator to generate the **phi** value from 0 - π;

```
theta = 2*np.pi*np.random.random(n)
phi = np.arccos(1 - 2*np.random.random(n))
```

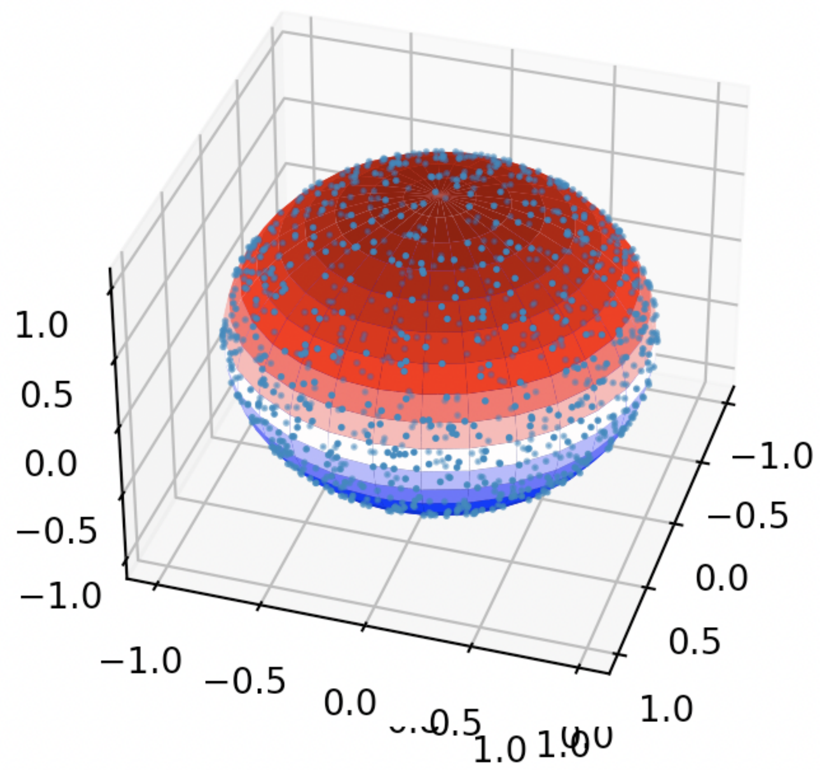
Since dA = dtheta * dphi*cos(phi),

If phi is distributing uniformly, the cos(phi) will not be uniform. I use inverse transformation sampling to make the dphi*cos(phi) uniform.

here we use `phi = np.arccos(1 - 2*np.random.random(n))` to make dphi cos(phi)part uniform.

The code is here

```
from matplotlib import pyplot as plt
#from mpl_toolkits.mplot3d import axes3d
import numpy as np

def sample_spherical(npoints, ndim=3):
    vec = np.random.randn(ndim, npoints)
    vec /= np.linalg.norm(vec, axis=0)
    return vec

plt.rcParams["figure.figsize"] = [7.00, 3.50]
plt.rcParams["figure.autolayout"] = True
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
r = 0.05
u, v = np.mgrid[0:2 * np.pi:30j, 0:np.pi:20j]
x = np.cos(u) * np.sin(v)
y = np.sin(u) * np.sin(v)
z = np.cos(v)


n = 1000

theta = 2*np.pi*np.random.random(n)
phi = np.arccos(1 - 2*np.random.random(n))

xs = np.cos(theta) * np.sin(phi)
ys = np.sin(theta) * np.sin(phi)
zs = np.cos(phi)
ax = fig.add_subplot(111, projection='3d')
ax.scatter(xs,ys,zs, s=1)
ax.plot_surface(x, y, z, cmap='seismic', alpha=1)
plt.show()
```



For problem b), I use 2 rectangles to show the Africa, and use a circle to show Antarctica.

Africa: (15’E to 40’W, 40’N to 0’) and (10’W to 40’W, 30’S to 0)

Antarctica: (69’S to 90’S)

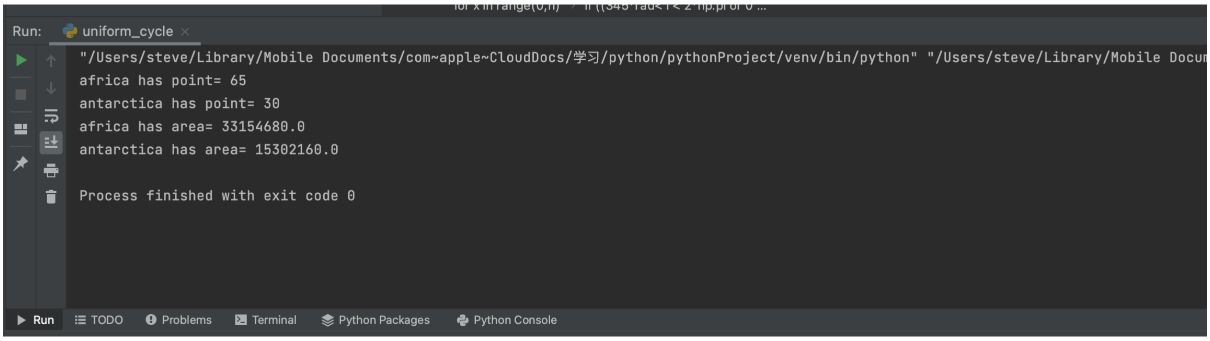
I sample 1000 points, and use the **Number** of point in these area / 1 * **Globe Area** = **Area**

```
africa_area = africa/n*globe_area
antarctica_area = antarctica/n*globe_area
```

Code this here:

```
from matplotlib import pyplot as plt
#from mpl_toolkits.mplot3d import axes3d
import numpy as np

def sample_spherical(npoints, ndim=3):
    vec = np.random.randn(ndim, npoints)
    vec /= np.linalg.norm(vec, axis=0)
    return vec

plt.rcParams["figure.figsize"] = [7.00, 3.50]
plt.rcParams["figure.autolayout"] = True
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
r = 0.05
u, v = np.mgrid[0:2 * np.pi:30j, 0:np.pi:20j]
x = np.cos(u) * np.sin(v)
y = np.sin(u) * np.sin(v)
z = np.cos(v)


n = 1000

theta = 2*np.pi*np.random.random(n)
phi = np.arccos(1 - 2*np.random.random(n))

africa = 0
```

```
antarctica = 0
globe_area = 510072000
#km^2

rad = np.pi/180
zr_lat = 1/2*np.pi

for x in range(0,n):
        i = theta[x]
        j = phi[x]
        if ((345*rad< i < 2*np.pi or 0 < i< 35*rad)and(zr_lat > j > zr_lat - 40*rad)):
            africa+=1
        if (10*rad<i<40*rad)and(zr_lat<j<zr_lat+30*rad):
            africa += 1
        if(zr_lat+69*rad < j < np.pi):
            antarctica+=1

africa_area = africa/n*globe_area
antarctica_area = antarctica/n*globe_area
print("africa has point=",africa)
print("antarctica has point=",antarctica)
print("africa has area=",africa_area)
print("antarctica has area=",antarctica_area)

xs = np.cos(theta) * np.sin(phi)
ys = np.sin(theta) * np.sin(phi)
zs = np.cos(phi)
ax = fig.add_subplot(111, projection='3d')
ax.scatter(xs,ys,zs, s=1)

ax.plot_surface(x, y, z, cmap='seismic', alpha=1)
plt.show()
```



Where is get 33.1 million Km^2 for Africa area and 15.3 Km^2 for Antarctica Area. These data are closed to the actual area

Africa / Area

30.37 million km²

Antarctica / **Area**

13.66 million km^2

Problem 2

Get a computer program for distinguishing a randomly generated sequence of zeroes and ones from a cooked-up one. You are welcome to write the program yourself or use what can you find on the web or in some book. Test your program on the following two sequences: the sequence consisting of the concatenation of all numbers in binary form1

0110111001011101111000...

and a similar sequence consisting of the concatenation of all prime numbers in binary form

0110111011111011 ...

The first sequence (the fractional part of the Champernowne number) is known to be random when considered in base 10; the second sequence (the fractional part of the Copeland-Erdös constant in binary form) is known to be random. In both cases, randomness is understood in a very specific way, and you are welcome to discuss this point too.

For this part, I find a test online by Stevenang on github.com. This program is cited from *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*

```
from numpy import zeros as zeros
from scipy.special import gammaincc as gammaincc
class Serial:

    @staticmethod
    def serial_test(binary_data:str, verbose=False, pattern_length=16):
        """
        From the NIST documentation http://csrc.nist.gov/publications/nistpubs/800-22-rev1a/SP800-22rev1a.pdf
        The focus of this test is the frequency of all possible overlapping m-bit patterns across the entire
        sequence. The purpose of this test is to determine whether the number of occurrences of the 2m m-bit
        overlapping patterns is approximately the same as would be expected for a random sequence. Random
        sequences have uniformity; that is, every m-bit pattern has the same chance of appearing as every other
        m-bit pattern. Note that for m = 1, the Serial test is equivalent to the Frequency test of Section 2.1.
        :param      binary_data:        a binary string
        :param      verbose             True to display the debug message, False to turn off debug message
        :param      pattern_length:     the length of the pattern (m)
        :return:    ((p_value1, bool), (p_value2, bool)) A tuple which contain the p_value and result of serial_test(True or False)
        """
        length_of_binary_data = len(binary_data)
        binary_data += binary_data[:(pattern_length -1):]

        # Get max length one patterns for m, m-1, m-2
        max_pattern = ''
        for i in range(pattern_length + 1):
            max_pattern += '1'

        # Step 02: Determine the frequency of all possible overlapping m-bit blocks,
        # all possible overlapping (m-1)-bit blocks and
        # all possible overlapping (m-2)-bit blocks.
        vobs_01 = zeros(int(max_pattern[0:pattern_length:], 2) + 1)
        vobs_02 = zeros(int(max_pattern[0:pattern_length - 1:], 2) + 1)
        vobs_03 = zeros(int(max_pattern[0:pattern_length - 2:], 2) + 1)

        for i in range(length_of_binary_data):
            # Work out what pattern is observed
            vobs_01[int(binary_data[i:i + pattern_length:], 2)] += 1
            vobs_02[int(binary_data[i:i + pattern_length - 1:], 2)] += 1
```

```
            vobs_03[int(binary_data[i:i + pattern_length - 2:], 2)] += 1

        vobs = [vobs_01, vobs_02, vobs_03]

        # Step 03 Compute for ψs
        sums = zeros(3)
        for i in range(3):
            for j in range(len(vobs[i])):
                sums[i] += pow(vobs[i][j], 2)
            sums[i] = (sums[i] * pow(2, pattern_length - i) / length_of_binary_data) - length_of_binary_data

        # Cimpute the test statistics and p values
        #Step 04 Compute for ∇
        nabla_01 = sums[0] - sums[1]
        nabla_02 = sums[0] - 2.0 * sums[1] + sums[2]

        # Step 05 Compute for P-Value
        p_value_01 = gammaincc(pow(2, pattern_length - 1) / 2, nabla_01 / 2.0)
        p_value_02 = gammaincc(pow(2, pattern_length - 2) / 2, nabla_02 / 2.0)

        if verbose:
            print('Serial Test DEBUG BEGIN:')
            print("\tLength of input:\t", length_of_binary_data)
            print('\tValue of Sai:\t\t', sums)
            print('\tValue of Nabla:\t\t', nabla_01, nabla_02)
            print('\tP-Value 01:\t\t\t', p_value_01)
            print('\tP-Value 02:\t\t\t', p_value_02)
            print('DEBUG END.')

        return ((p_value_01, p_value_01 >= 0.01), (p_value_02, p_value_02 >= 0.01))
```

This program focus on the occurrences of m-bits sub-string in the binary string. The purpose of this test is to determine whether the number of occurrences of the 2m m-bits overlapping patterns is approximately the same as would be expected for a random sequence. For a random string, occurrences of every m-bits pattern is the same.

Running the code:

```
import Distinguish_machine as DS

tst = DS.Serial
print(tst.serial_test("0110111001011101111000"))

print(tst.serial_test("0110111011111011"))
```

It turns out that, both of the string are random.