MATH407 Computer Project

Jacob Ma

November 30, 2022

Problem 1

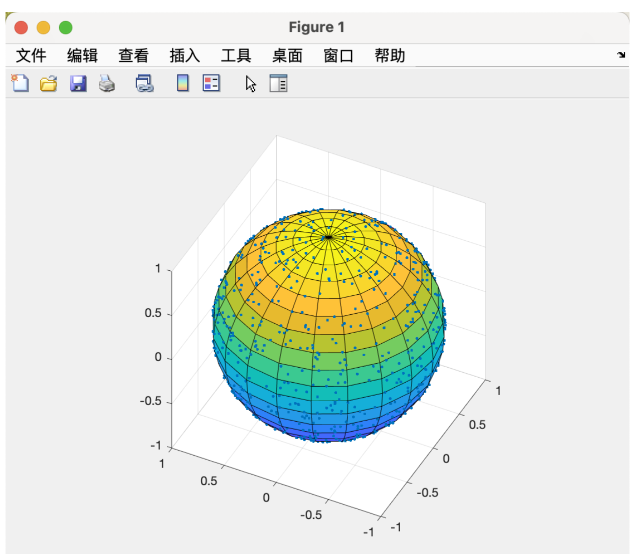
Find a procedure for sampling uniformly on the surface of the sphere.

(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.)

Solution

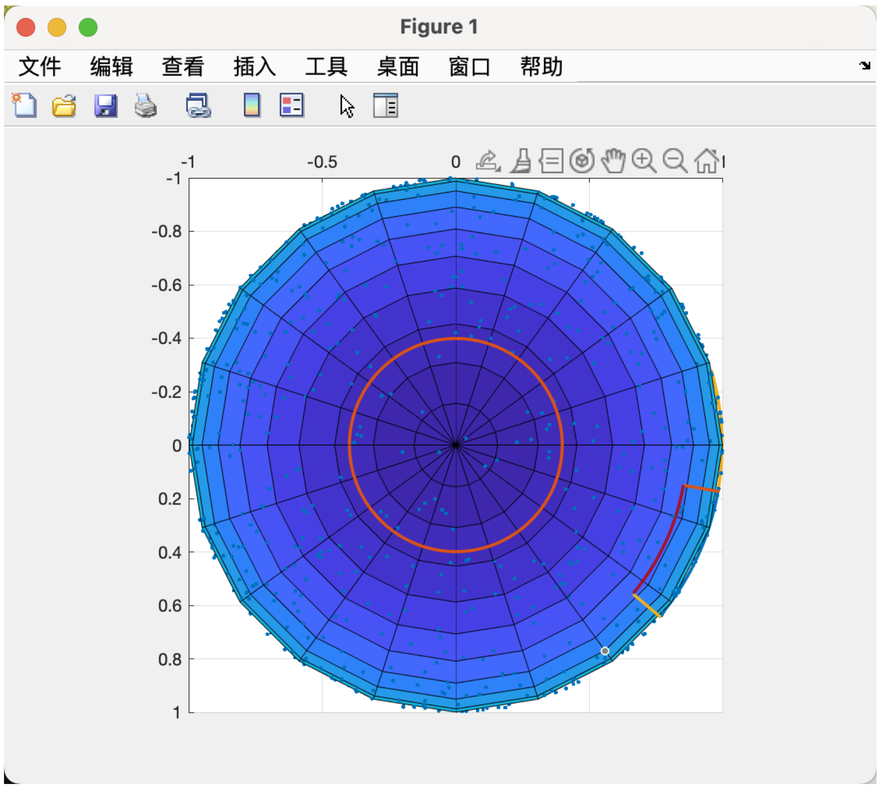
Here is some screenshots of the image generated by MatLab.



Drawing 1000 points that are random, independent, and uniform on the unit sphere

And as for problem $1\,(b)$, I abstract Antarctica into a disk and Africa into two rectangles. By calculating how many points are within the disk / rectangle, we can calculate how many percent of the points lands in the Antarctica and Africa area, thus calculating their area by multiplying the surface area of Earth, which is 510067866 km ^{2}.

Here is how I calculate the area of Antarctica:



Abstraction of Antarctica

```
>> MATH407_CP1_
    30
```

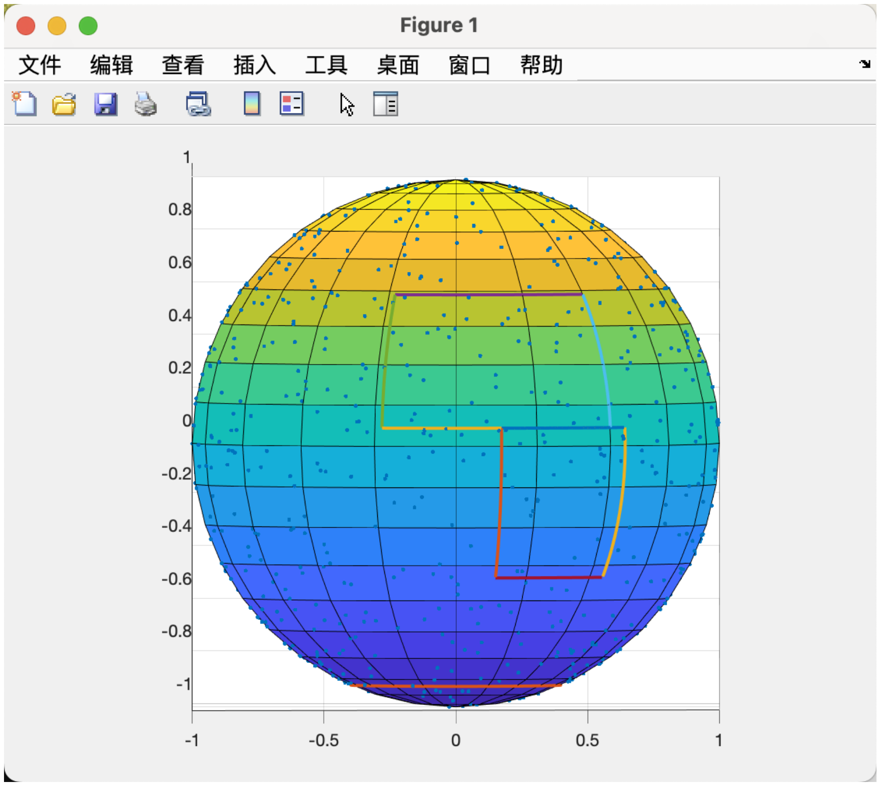
The console prints out that there are 30 points in this area

Thus, we can estimate the are of Antarctica

$$
\begin{aligned}
A\,(Antarctica) &\approx \frac{30}{1000} \cdot 510067866 \\
&\approx 15302035
\end{aligned}
$$

This answer has 9.3% relative error from the actual value 14000000.

Similarly, here is how I calculate the area of Africa



Abstraction of Africa

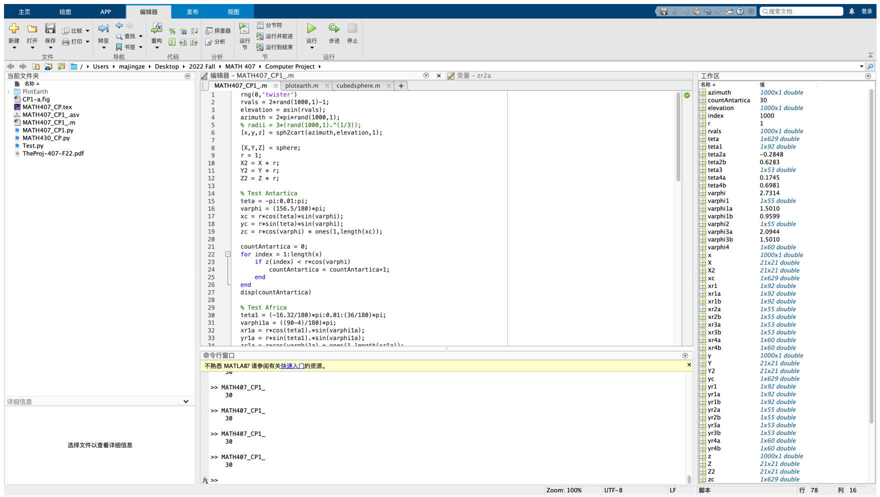
There are 65 points within the area. Thus, we can approximate the area

$$\begin{aligned} A\,(Africa) &\approx \frac{65}{1000} \cdot 510067866 \\ &\approx 33154411 \end{aligned}$$

Compared to the actual value $30,221,532$, there are 9.7% of relative error.

As we can observe, the relative error is quite similar in this case, and both within a reasonable range but not close enough. This could be because of how to abstract the shape of both continents and how we define the area of these two continents.

Here is a screenshot of my code:



MatLab Screenshot

And the code is as follows in detail if you would like to read

```
rng(0,'twister')
rvals = 2*rand(1000,1)-1;
elevation = asin(rvals);
azimuth = 2*pi*rand(1000,1);
% radii = 3*(rand(1000,1).^(1/3));
[x,y,z] = sph2cart(azimuth,elevation,1);

[X,Y,Z] = sphere;
r = 1;
X2 = X * r;
Y2 = Y * r;
Z2 = Z * r;

% Test Antartica
teta = -pi:0.01:pi;
varphi = (156.5/180)*pi;
xc = r*cos(teta)*sin(varphi);
yc = r*sin(teta)*sin(varphi);
zc = r*cos(varphi) * ones(1,length(xc));

countAntartica = 0;
for index = 1:length(x)
    if z(index) < r*cos(varphi)
        countAntartica = countAntartica+1;
    end
end
disp(countAntartica)

% Test Africa
teta1 = (-16.32/180)*pi:0.01:(36/180)*pi;
varphi1a = ((90-4)/180)*pi;
```

```
xr1a = r*cos(teta1).*sin(varphi1a);
yr1a = r*sin(teta1).*sin(varphi1a);
zr1a = r*cos(varphi1a) * ones(1,length(xr1a));

varphi1b = ((90-35)/180)*pi;
xr1b = r*cos(teta1).*sin(varphi1b);
yr1b = r*sin(teta1).*sin(varphi1b);
zr1b = r*cos(varphi1b) * ones(1,length(xr1b));


varphi2 = (55/180)*pi:0.01:(86/180)*pi;
teta2a = (-16.32/180)*pi;
xr2a = r*cos(teta2a).*sin(varphi2);
yr2a = r*sin(teta2a).*sin(varphi2);
zr2a = r*cos(varphi2);

teta2b = (36/180)*pi;
xr2b = r*cos(teta2b).*sin(varphi2);
yr2b = r*sin(teta2b).*sin(varphi2);
zr2b = r*cos(varphi2);

% Lower Part of Africa
teta3 = (10/180)*pi:0.01:(40/180)*pi;
varphi3a = ((90+30)/180)*pi;
xr3a = r*cos(teta3).*sin(varphi3a);
yr3a = r*sin(teta3).*sin(varphi3a);
zr3a = r*cos(varphi3a) * ones(1,length(xr3a));

varphi3b = ((90-4)/180)*pi;
xr3b = r*cos(teta3).*sin(varphi3b);
yr3b = r*sin(teta3).*sin(varphi3b);
zr3b = r*cos(varphi3b) * ones(1,length(xr3b));


varphi4 = ((90-4)/180)*pi:0.01:((90+30)/180)*pi;
teta4a = (10/180)*pi;
xr4a = r*cos(teta4a).*sin(varphi4);
yr4a = r*sin(teta4a).*sin(varphi4);
zr4a = r*cos(varphi4);

teta4b = (40/180)*pi;
xr4b = r*cos(teta4b).*sin(varphi4);
yr4b = r*sin(teta4b).*sin(varphi4);
zr4b = r*cos(varphi4);

figure
plot3(x,y,z,'.')
hold on
plot3(xc,yc,zc,'linewidth',2)
plot3(xr1a,yr1a,zr1a,'linewidth',2)
plot3(xr1b,yr1b,zr1b,'linewidth',2)
```

```
plot3(xr2a,yr2a,zr2a,'linewidth',2)
plot3(xr2b,yr2b,zr2b,'linewidth',2)
plot3(xr3a,yr3a,zr3a,'linewidth',2)
plot3(xr3b,yr3b,zr3b,'linewidth',2)
plot3(xr4a,yr4a,zr4a,'linewidth',2)
plot3(xr4b,yr4b,zr4b,'linewidth',2)
surf(X2,Y2,Z2)
axis equal
% plotearth()
grid on
hold off
```

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 form

$$0110111001011101111000...$$

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

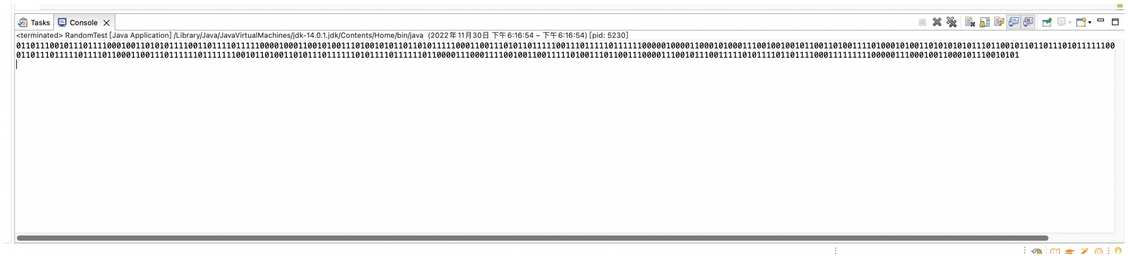
$$0110111011111011...$$

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-Erdos 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.

Solution

For convince, I used Java to generate these two binary sequence, and an existing Python method to distinguish their randomness.

As for generating random numbers, I am generating from $1 - 50$ for the first sequence, and from $1 - 150$ for the prime binary sequence to make them both have enough length to enter the test. Here is the screenshot of my code and output.



Screen shot of Out Put

The output are as follows in text way

011011100101110111100010011010101111001101111011111000010···

01101110111111011110110001100111011111110111111100101101001101···

Comparing to the sequence given, I believe this is a correct output.

The actual Java code is as follows, you can skip this if you do not need to read this

```java
package jingzema_MATH407_CP;

public class RandomTest {

   public static void main(String[] args) {
      String s1 = generateBinaryA();
      String s2 = generateBinaryB();
      System.out.println(s1);
      System.out.println(s2);
   }

   public static String generateBinaryA() {
      String result = "";
      for (int i=0; i<=50; i++) {
         result+=toBinary(i);
      }
      return result;
   }

   public static String generateBinaryB() {
      String result = "";
      for (int i=0; i<=150; i++) {
         if (checkPrime(i)) {
            result+=toBinary(i);
         }
      }
      return result;
   }

   public static String toBinary(int decimal){
        int binary[] = new int[40];
```

```
        int index = 0;
        String result = "";
        if (decimal == 0) {
          return "0";
        }
        while(decimal > 0){
          binary[index++] = decimal%2;
          decimal = decimal/2;
        }
        for(int i = index-1;i >= 0;i--){
          result+=binary[i];
        }
        return result;
    }

    public static boolean checkPrime(int n){
        int i,m=0;
        m=n/2;
        if(n==0||n==1) {
           return true;
        }
        for(i=2;i<=m;i++) {
           if(n%i==0){
              return false;
           }
        }
        return true;
    }
}
```

As for the randomness test, we are applying Serial Test from $2-26$ from **A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications**, and code from *Steven Wang* on GitHub.

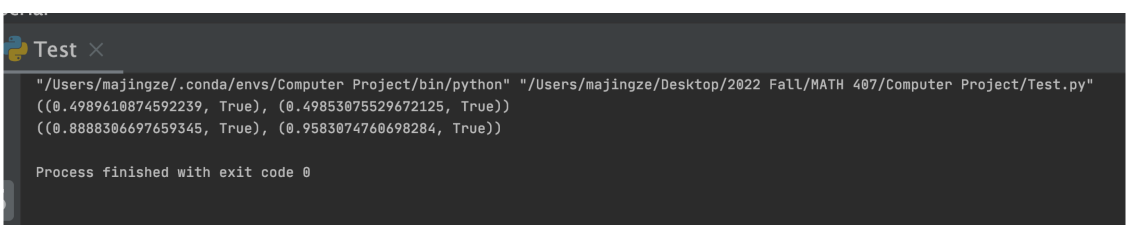
The main idea is that testing the frequency of all possible overlapping m-bit patterns across the entire sequence. The purpose of this test is to determine whether the number of occurrences of the 2m m-bit overlapping patterns is approximately the same as would be expected for a random sequence. Random sequences have uniformity; that is, every m-bit pattern has the same chance of appearing as every other m-bit pattern. (Andrew, $2-26$), the code is as follows

```
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 s
        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 Delta
        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:
```

```python
            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))

    print(serial_test("0110111001011101111000100110101011110011011110111110000100011001010011101001010110110101111100011001110
    print(serial_test("0110111011111101111011000110011101111110111111100101101001101011101111110101111011111101100001110001110
```

And we can see the results in the shell/terminal as following



Random Test Terminal Output

And both of them shows to be random. Thus done.