PrimeVizProject+(Revised)

December 13, 2018

1 Report 2: Visualizing the prime numbers

1.1 By Nicolas Madhavapeddy, Alexander Soe, Shahin Karami, Joseph Ismailyan, and Daniel Sanchez

2 Introduction

By definition, a prime number is a whole number whose only factors are one and itself. Because of this unique property, primes have been continuously studied throughout history, even as far as the ancient Greeks. And yet even after all this time, why do we continue to study them? It is because there are so many unique properties and discoveries that only the primes exhibit. In fact, we can construct the whole number line with nothing but primes. Mathematicians over the years have discovered these properties over many centuries and in turn have found many ways to represent primes. One of them is through visualization. In this notebook, we use Ulam's spiral and a histogram of the distribution of the primes that are equivalent to the sum of two squares. In each of these visualizations, we explore the behavior of primes and the patterns that arise when represented in that manner. We used PIL for Ulam's spiral and matplotlib for graphing sums of squares that is prime.

3 Prime Visualization: Ulam's Spiral

```
In [1]: from PIL import Image
        import numpy as np
```

This is a modified version of `Image.fromarray(...)`. It changes the x and y axis. This goes from bottom-up instead of top-down.

```
In [2]: def array_to_img(pixarray):
            '''
            Makes a PIL image from a pixel array.
            '''
            newarray = np.swapaxes(pixarray[:,-1::-1,:],0,1)
            # Switches x and y axis.  Goes bottom-up instead of top-down too.
            return Image.fromarray(newarray)
```

`is_prime(n)` is a helper function that checks if the argument `n` is prime. It returns `True` if prime and `False` if it is not prime.

```
In [3]: import math
        def is_prime(n):
            '''
            Checks whether the argument n is a prime number.
            Uses a brute force search for factors between 1 and sqrt(n).
            '''
            if n == 1:
                return False
            if n == 2:
                return True
            if n%2 == 0:
                #print("{} is a factor of {}.".format(2,n))
                return False
            j = 3
            root_n = math.sqrt(n)
            while j <= root_n:
                # j will proceed through the list of odd numbers 3,5,... up to sqrt(n).
                if n%j == 0:  # is n divisible by j?
                    #print("{} is a factor of {}.".format(j,n))
                    return False
                j = j + 2  # There's a Python abbreviation for this:  j += 2.
            return True
```

It is famously said that the mathematician Stanislaw Ulam came up with the spiral during a boring lecture at a scientific presentation. Ulam created the spiral by placing one in the center with the primes circling around one.

The cells below execute Ulam's spiral. We start the spiral at the bottom right pixel where the entry is `image_size*image_size`. Note that entry is the last and largest number in the spiral. Then we start to move to the left and at the same time check if the number corresponding to that pixel is prime. If it is prime, then that pixel is changed to a non-black color. When the left direction has reached the end of that row, then it moves in the up direction along the column. Once the up direction reaches the boundary, it proceeds to move right. The final direction is downwards. It continues to move left, up, right, and down until we reach the pixel that corresponds to the number one. Ultimately, we are starting the spiral from the boundary of the image and then spiral into the center.

We need to define the size of the image that our spiral will take place. It should be an odd number because an odd number by an odd number has a center. Let it be a 101 by 101 image.

```
In [4]: img_size = 101
```

`pixels` is an array that we manipulate and then is converted into an image.

```
In [5]: pixels = np.zeros((img_size,img_size,3),dtype=np.uint8)
```

`flag` is set to True to continue the left, up, right, and down direction and is used for the `while` loop condition.

`row1` is used to help the left direction by accessing the pixels at row1. This means that the row is constant and the columns are changing.

`column1` is used to help the up direction by accessing the pixels at column1. It serves the same purpose as `row1`, but in this case, column is constant and the rows are changing.

`boundary` is used to eliminate the pixels that have already been processed

```
In [6]: flag = True
        row1 = 0
        column1 = 0
        boundary = 0
```

The list `nums` contains the numbers that correspond to each pixel and will determine if that pixel is a non-black color or not. The list starts with the largest number, decreases by 1 and stops at one. The variable `n` is used to access the elements of `nums`.

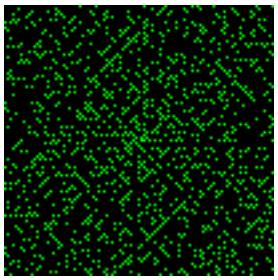
```
In [7]: nums = np.array( list(range(img_size*img_size,0,-1)) )
        n = 0
```

Every *for loop* is a movement of either up, down, left or right. These *for loops* are in a *while loop* to make the movements continuous until we have reached the pixel that is associated with the number one. The first *for loop* is the left direction and is the last direction to execute because we approach one in the left direction. The second *for loop* is the up direction. The third *for loop* is the right direction and the last is the down direction.

```
In [8]: while flag != False:

            for i in range(-1-boundary, -img_size-1+boundary ,-1):
                if is_prime(nums[n]):
                    pixels[i,row1] = [0,255,0]
                if nums[n] == 1:
                    #this if statment checks if we have arrived at the center and sets
                    flag = False
                    #so it can end the for loop and not run into an index
                    #error
                    continue
                n+=1    #n is incremented in every for loop because every pixel is associated
                        #to a number

            if flag == False:
                #if nums[n] == False is true then we are at the center and have to exit
                #the while loop
                    continue



            for j in range(1+boundary,img_size-boundary):
                if is_prime(nums[n]):
                    pixels[column1,j] = [0,255,0]
                n+=1
```

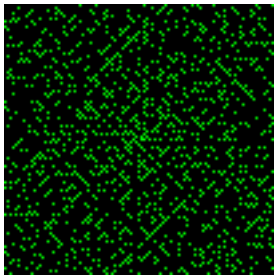
```
            row2 = img_size - 1 - boundary

            for i in range(1 + boundary,img_size - boundary):
                if is_prime(nums[n]):
                    pixels[i,row2] = [0,255,0]
                n+=1

            column2 = img_size - 1 - boundary

            for j in range(-2 - boundary, -img_size + boundary, -1):
                if is_prime(nums[n]):
                    pixels[column2,j] = [0,255,0]
                n+=1

            #we increment boundary and row1 and column1 to eliminate
            #the pixels that have been changed to another color
            boundary +=1
            row1 +=1
            column1 +=1
```

```
In [9]: img = array_to_img(pixels)
        display(img)
```



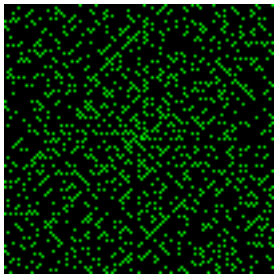
We can see that there are patterns in the image above. There are diagonal lines that have many primes. It is known that there are quadractic equations that produce these diagonal lines such as Euler's prime generating function, $f(n) = n^2 - n + 41$. In order to accomplish this, we will have 41 in the center by adding 40 to each element of `nums` and end at 41 instead of 1. The distinctive diagonal line in the image below are values of Euler's function. This tells us that there are functions that produce a high frequency of primes.

```
In [10]: eulerLine = array_to_img(pixels)
         display(eulerLine)
```



We then started to think about how the numbers behave mod n. We took the image above and wanted to see how the pixel's change in color with mod 4. In our image, 1 mod 4 is cyan and 3 mod 4 is yellow.

```
In [11]: eulerLine4 = array_to_img(pixels)
         display(eulerLine4)
```



Euler's diagonal line alternates between cyan and yellow. However, what really got us stuck were the diagonal lines that have negative slopes. Each line either has only cyan primes or only yellow primes. Moreover, the cyan and yellow lines alternate. When we look at the positive slope lines, they also alternate between cyan primes and yellow primes but on the same line which is explained by the negative slope lines and how the yellow and blue lines alternate. This behavior is consistent with any center. A possible conjecture is that if a negative slope diagonal line has a prime that is 1 mod 4, then all the primes on that diagonal line will be 1 mod 4. The same can be said about the primes that are 3 mod 4.

4 Prime Visualization: Sum of Two Squares

This is just the starter code for the Sieve of Eratosthenes except that we added an additional for-loop at the bottom that converts the list of boolean values to just a list of primes. Since primes don't necessarily have any definitive patterns in terms of how often they appear, we have to create a list of primes to look through in order to check if they satisfy the given criteria of being a sum of two squares. If we were to check if each number was a prime as well as a prime that was a sum of two squares separately in each function, then the runtime would be exponentially longer as we would have to nest another for-loop. So all in all, having a list of primes ready to go will help our code run faster.

```
In [12]: #Sieve of Eratosthenes
         import math
         def isprime_list(n):
             '''
             Return a list of length n+1
             with Trues at prime indices and Falses at composite indices.
             '''
             primes = []
             flags = [True] * (n+1)  # A list [True, True, True,...] to start.
             flags[0] = False  # Zero is not prime.  So its flag is set to False.
             flags[1] = False  # One is not prime.  So its flag is set to False.
             flags[4::2] = [False] * ((n-2)//2)
             p = 3
             while p <= math.sqrt(n):  # We only need to sieve by p is p <= sqrt(n).
                 if flags[p]:  # We sieve the multiples of p if flags[p]=True.
                     flags[p*p::2*p] = [False] * ((n-p*p-1)//(2*p)+1) # Sieves out multiples o
                 p = p + 2 # Try the next value of p.
             for j in range(n):   #Loops through the list and converts the boolean values to i
                 if flags[j]:
                     primes.append(j)
             return primes   #Returns a List of primes from 1-n
```

This snippet of code was my first attempt at attacking the problem. We simply ran a triple for-loop where the first two loop through all the values between 0 and n, and the third nested loop checks if any of the primes in the list are a sum of the outer loop's values squared. Later we found out that we can use "in" to check if the sum of squares was "in" the prime list. At first we had the second loop incrementing from 0 but we noticed that we were getting repeated values and it was taking a significant amount of time. We realized that by changing the initial starting place of the second loop to be equal to the first loops current place, we could speed up the runtime as well as get rid of unnecessary repeating primes. This makes sense as my older version would see 1^2 + 2^2 = 2^2 + 1^2 = 5, which is a prime, and therefore print it twice. This current version skips the second check. Lastly, as in all the other pieces of code we printed the number of primes as well as the actual list of primes. If you try to run this code it will take a couple of minutes before it returns the primes. In the next example the same list will be achieved almost instantly.

```
In [13]: #Brute Force method for first part. Finds all the primes that are a sum of squares.
         import numpy as np
         def sumOfSquares(n):
             primes = np.array(isprime_list(n))
             sum = []
             for i in range(0, n):
                 for j in range(i, n):
                     for k in range(len(primes)):
                         if primes[k] == (i**2 + j**2):
                             sum.append(primes[k])
             result = np.array(sorted(sum))
             print("Number of Primes = ",len(result))
             return (result)
         sumOfSquares(1000)
```

```
Number of Primes =  81
```

```
Out[13]: array([  2,   5,  13,  17,  29,  37,  41,  53,  61,  73,  89,  97, 101,
                109, 113, 137, 149, 157, 173, 181, 193, 197, 229, 233, 241, 257,
                269, 277, 281, 293, 313, 317, 337, 349, 353, 373, 389, 397, 401,
                409, 421, 433, 449, 457, 461, 509, 521, 541, 557, 569, 577, 593,
                601, 613, 617, 641, 653, 661, 673, 677, 701, 709, 733, 757, 761,
                769, 773, 797, 809, 821, 829, 853, 857, 877, 881, 929, 937, 941,
                953, 977, 997])
```

This is a modified version that runs a lot faster as it omits the third inner for-loop.

```
In [14]: #Faster version of the first case that doesn't use Fermat' theorem.
         #Finds all the primes that are a sum of squares.
         import numpy as np
         def sumOfSquares(n):
             primes = np.array(isprime_list(n))
             sum = []
             for i in range(0, n):
                 for j in range(i, n):
                     sumOfSquares = i**2 + j**2
                     if (sumOfSquares) in primes:
                         sum.append(sumOfSquares)
             result = np.array(sorted(sum))
             print("Number of Primes = ",len(result))
             return (result)
         sumOfSquares(1000)
```

```
Number of Primes =  81
```

```
Out[14]: array([  2,   5,  13,  17,  29,  37,  41,  53,  61,  73,  89,  97, 101,
                109, 113, 137, 149, 157, 173, 181, 193, 197, 229, 233, 241, 257,
                269, 277, 281, 293, 313, 317, 337, 349, 353, 373, 389, 397, 401,
                409, 421, 433, 449, 457, 461, 509, 521, 541, 557, 569, 577, 593,
                601, 613, 617, 641, 653, 661, 673, 677, 701, 709, 733, 757, 761,
                769, 773, 797, 809, 821, 829, 853, 857, 877, 881, 929, 937, 941,
                953, 977, 997])
```

After doing some research and note checking from our previous number theory course, we remembered that Fermat's sum of two squares theorems stated that "a prime is a sum of two squares iff it is congruent to 1(mod 4). Additionally "a prime is a sum of a square and twice a square iff it is congruent to 1(mod 8) or 3(mod 8)" and "a prime is a sum of a square and three times a square iff it is congruent to 1(mod 3)". Therefore by using these theorems, we can drastically reduce runtime by ommiting the the two outer for-loops from the original code and replace them with just a conditional statement. This in turn gives us the same exact result as the previous code snippet but substantially faster runtime. This also makes it possible for us to easily compute n = 1,000,000.

```
In [15]: #Fermat's theorem on first part. Finds all the primes that are a sum of squares.
         import numpy as np
         def sumOfSquaresFaster(n):
             sumFaster = [2]
             primes = isprime_list(n)
             for i in range(len(primes)):
                 if(primes[i] % 4 == 1):
                     sumFaster.append(primes[i])
             result = np.array(sumFaster)
             print("Number of Primes = ",len(result))
             return (result)
         sumOfSquaresFaster(1000000)

Number of Primes =  39176


Out[15]: array([     2,      5,     13, ..., 999917, 999953, 999961])
```

Even though Fermat's sum of two squares theorems are much easier to implement, we thought it would be beneficial to write the brute force method for each case just to help with debugging and to make sure that both functions were creating the same output. Again each brute force method is still super inefficient. One key thing to note after each snippet of code that uses Fermat's theorem is that we convert the final list into a numpy array. This is super important for the last function.

```
In [16]: #Brute Force method for second part. Finds all the primes that are a sum of a square
         import numpy as np
         def sumOfSquares2(n):
             primes = isprime_list(n)
             sum2 = []
             for i in range(0, n):
                 for j in range(0, n):
                     temp = 2 * i**2 + j**2
                     if (temp) in primes:
                         sum2.append(temp)
             result2 = np.array(sorted(sum2))
             print("Number of Primes = ",len(result2))
             return(result2)
         sumOfSquares2(1000)

Number of Primes =  82


Out[16]: array([  2,   3,  11,  17,  19,  41,  43,  59,  67,  73,  83,  89,  97,
                107, 113, 131, 137, 139, 163, 179, 193, 211, 227, 233, 241, 251,
                257, 281, 283, 307, 313, 331, 337, 347, 353, 379, 401, 409, 419,
                433, 443, 449, 457, 467, 491, 499, 521, 523, 547, 563, 569, 571,
                577, 587, 593, 601, 617, 619, 641, 643, 659, 673, 683, 691, 739,
                761, 769, 787, 809, 811, 827, 857, 859, 881, 883, 907, 929, 937,
                947, 953, 971, 977])
```

In [17]:
```
#Fermat's theorem on second part. Finds all the primes that are a sum of a square and
def sumOfSquares2Faster(n):
    sum2Faster = [2]
    primes = isprime_list(n)
    for i in range(len(primes)):
        if(primes[i] % 8 == 1 or primes[i] % 8 == 3):
            sum2Faster.append(primes[i])
    result2 = np.array(sum2Faster)
    print("Number of Primes = ",len(result2))
    return(result2)
sumOfSquares2Faster(1000000)
```

```
Number of Primes =  39206
```

Out[17]:
```
array([     2,      3,     11, ..., 999953, 999961, 999979])
```

In [18]:
```
#Brute Force method for third part. Finds all the primes that are a sum of a square a
def sumOfSquares3(n):
    primes = isprime_list(n)
    sum3 = []
    for i in range(0, n):
        for j in range(0, n):
            temp = 3 * i**2 + j**2
            if (temp) in primes:
                sum3.append(temp)
    result3 = np.array(sorted(sum3))
    print("Number of Primes = ",len(result3))
    return(result3)
sumOfSquares3(1000)
```

```
Number of Primes =  81
```

Out[18]:
```
array([  3,   7,  13,  19,  31,  37,  43,  61,  67,  73,  79,  97, 103,
       109, 127, 139, 151, 157, 163, 181, 193, 199, 211, 223, 229, 241,
       271, 277, 283, 307, 313, 331, 337, 349, 367, 373, 379, 397, 409,
       421, 433, 439, 457, 463, 487, 499, 523, 541, 547, 571, 577, 601,
       607, 613, 619, 631, 643, 661, 673, 691, 709, 727, 733, 739, 751,
       757, 769, 787, 811, 823, 829, 853, 859, 877, 883, 907, 919, 937,
       967, 991, 997])
```

In [19]:
```
#Fermat's Theorem for third part. Finds all the primes that are a sum of a square and
def sumOfSquares3Faster(n):
    sum3Faster = []
    primes = isprime_list(n)
    sum3Faster.append(3)
    for i in range(len(primes)):
        if(primes[i] % 3 == 1):
```

```
                sum3Faster.append(primes[i])
        result3 = np.array(sum3Faster)
        print("Number of Primes = ",len(result3))
        return(result3)
    sumOfSquares3Faster(1000000)
```

```
Number of Primes =  39232
```

```
Out[19]: array([     3,      7,     13, ..., 999931, 999961, 999979])
```

After running each case of Fermat's theorem on the list of primes from 1 to 1,000,000, we can now try to find any patterns with the frequencies of ending digits. Since we were able to return a numpy array for each function, all we have to do to convert the prime arrays into ending digits is to apply "% 10" to the entire array. Then to find the frequencies we just add up the number of primes that end in 1, 3, 7, and 9 and return them.

```
In [20]: #Finds the Frequencies of last digits.
         def lastDigitFreq(n):
             num1 = 0
             num3 = 0
             num7 = 0
             num9 = 0
             n = n % 10
             for i in range(len(n)):
                 if(n[i] == 1):
                     num1 += 1
                 if(n[i] == 3):
                     num3 += 1
                 if(n[i] == 7):
                     num7 += 1
                 if(n[i] == 9):
                     num9 += 1
             print("\n Number of primes ending in 1 =", num1,"\n","Number of primes ending in 3
```

For the most part, the distribution of primes is pretty tight; however primes ending in 3 or 7 tend to be slightly more frequent than ones that end in 1 or 9 if the prime is a sum of two squares. It seems that in comparison to the second and third case, the first case has the largest gap between ending digit frequencies with there being 74 more primes that end in both 3 and 7 than in 9. Additionally this case in particular has the least amount of primes in a sample size of 1,000,000.

```
In [21]: #Case 1 (p = i^2 + j^2)
         lastDigitFreq(sumOfSquaresFaster(1000000))
```

```
Number of Primes =  39176

 Number of primes ending in 1 = 9758
 Number of primes ending in 3 = 9830
 Number of primes ending in 7 = 9830
```

```
 Number of primes ending in 9 = 9756
```

The last digit frequency gap for the sum of a square and twice a square is significantly smaller in this example with it being almost half of the last. There are only 38 more primes that end in 3 than primes that end in 9. There is also a larger amount of primes that fulfill case 2 than case 1.

```
In [22]: #Case 2 (p = 2*i^2 + j^2)
         lastDigitFreq(sumOfSquares2Faster(1000000))

Number of Primes =  39206

 Number of primes ending in 1 = 9795
 Number of primes ending in 3 = 9818
 Number of primes ending in 7 = 9812
 Number of primes ending in 9 = 9780
```

Lastly, the last digit frequency gap for the sum of a square and three times is also pretty small in this example but not as drastic as comparison between case 1 and 2. There are only 37 more primes that end in 3 than primes that end in 9. Thats only a single prime difference. There is also a larger amount of primes that fulfill case 3 than in either case 1 or 2.

```
In [23]: #Case 3 (p = 3*i^2 + j^2)
         lastDigitFreq(sumOfSquares3Faster(1000000))

Number of Primes =  39232

 Number of primes ending in 1 = 9807
 Number of primes ending in 3 = 9825
 Number of primes ending in 7 = 9812
 Number of primes ending in 9 = 9788
```

For the visualization of these programs it is necassary to slightly alter the programs above to return values that are useful for graphing. It is important to note that the bulk of the programs remain the same, only a few things around the edges have been altered. First we will alter sumOfSquares so that we can plot the results on a scatter plot.

```
In [24]: #Brute Force method for first part.
         import numpy as np
         def sumOfSquares_plot(n):
             primes = np.array(isprime_list(n))
             sum = []
             x_values = []
             y_values = []
```

```
            for i in range(0, n):
                for j in range(i, n):
                    temp = i**2 + j**2
                    if temp in primes :
                        x_values.append(i)
                        y_values.append(j)
                        sum.append(temp)
            result = np.array(sorted(sum))
            return (x_values, y_values)
```

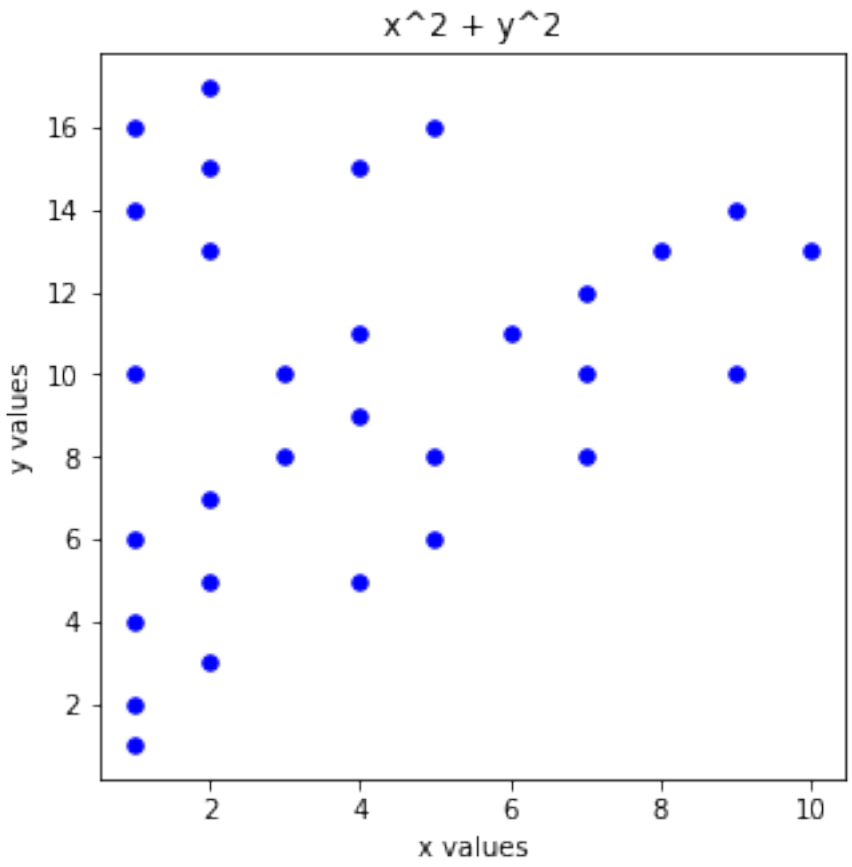
```
In [25]: sumOfSquares_plot(100)
```

```
Out[25]: ([1, 1, 1, 1, 2, 2, 2, 3, 4, 4, 5, 5], [1, 2, 4, 6, 3, 5, 7, 8, 5, 9, 6, 8])
```

Here we simply added two lists called x_values and y_values to the program. This is in order to keep track of which x and y values satisfy x^2 + y^2 = p for some prime p. The Function sumOfSquares_plot then returns these two lists.

```
In [26]: import matplotlib.pyplot as plt # Use plt.<command> hereafter.
```

```
In [27]: xlist =  sumOfSquares_plot(300)[0] # A list of 500 random floats between 0 and 1.
         ylist = sumOfSquares_plot(300)[1] # Another such list.

         plt.figure(figsize=(5,5))
         plt.scatter(xlist, ylist, alpha=1, c='blue', s=30)
         plt.xlabel('x values')
         plt.ylabel('y values')
         plt.title('x^2 + y^2')
         plt.show()
```



It is important to note that there appears to be a lower bound as x increases. This is an illusion of the program however. Take 29 (prime) for example. In the case of 29, x = 2 and y = 5 and this is represented on the graph. The similar case of x = 5 and y = 2 is not represesnted on the graph because in our program it is considered a trivial case and not considered.

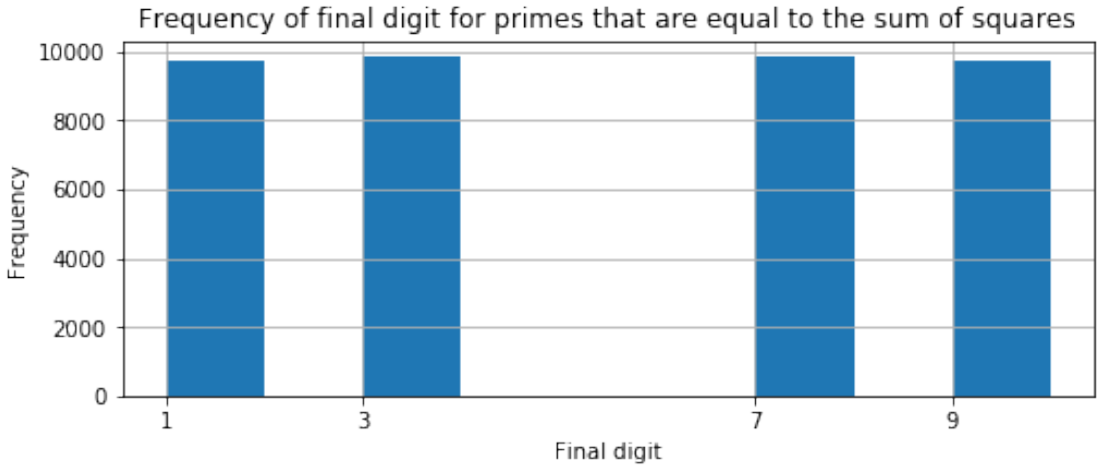
Next the lastDigitFreq function is altered in order to represent the values in a histogram. We therefore need to return a count of every prime between 1 and n that satisfies the condition and ends in 1, 3, 7, and 9. Therefore every time we find a prime that ends in nine, we append a nine to the list (and similarly for numbers 1, 3, and 7).

```
In [28]: #Finds the Frequencies of last digits
         def lastDigitFreq_plot(n):
             num = []
             n = n % 10
             for i in range(len(n)):
                 if(n[i] == 1):
                     num.append(1)
                 if(n[i] == 3):
                     num.append(3)
                 if(n[i] == 7):
                     num.append(7)
```

```
            if(n[i] == 9):
                num.append(9)
        return(num)
```

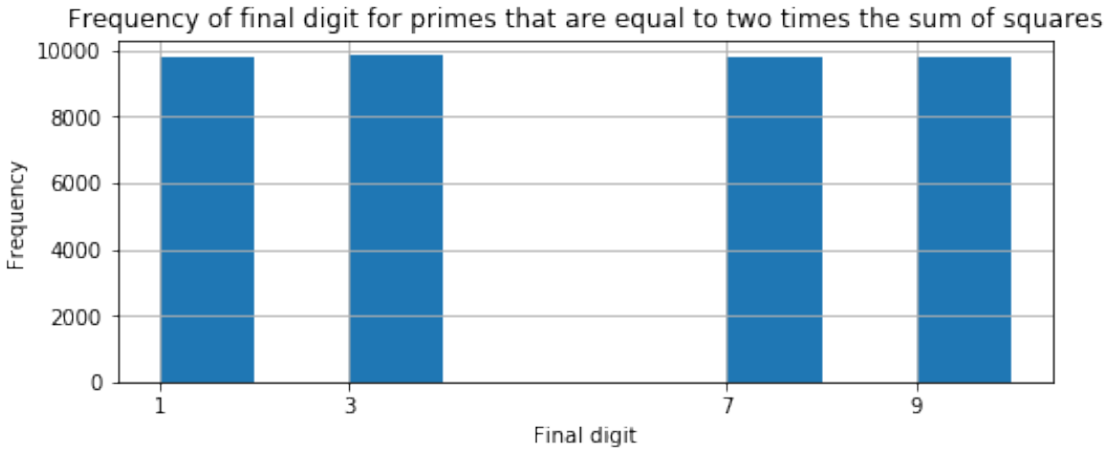
```
In [29]: %matplotlib inline
         plt.figure(figsize=(8, 3))  #  Makes the resulting figure 12in by 5in.
         plt.hist(lastDigitFreq_plot(sumOfSquaresFaster(1000000)), bins=range(1,11)) #  Makes
         plt.ylabel('Frequency')
         plt.xlabel('Final digit')
         plt.xticks([1,3,7,9])
         plt.grid(True)
         plt.title('Frequency of final digit for primes that are equal to the sum of squares')
         plt.show()
```

```
Number of Primes =  39176
```

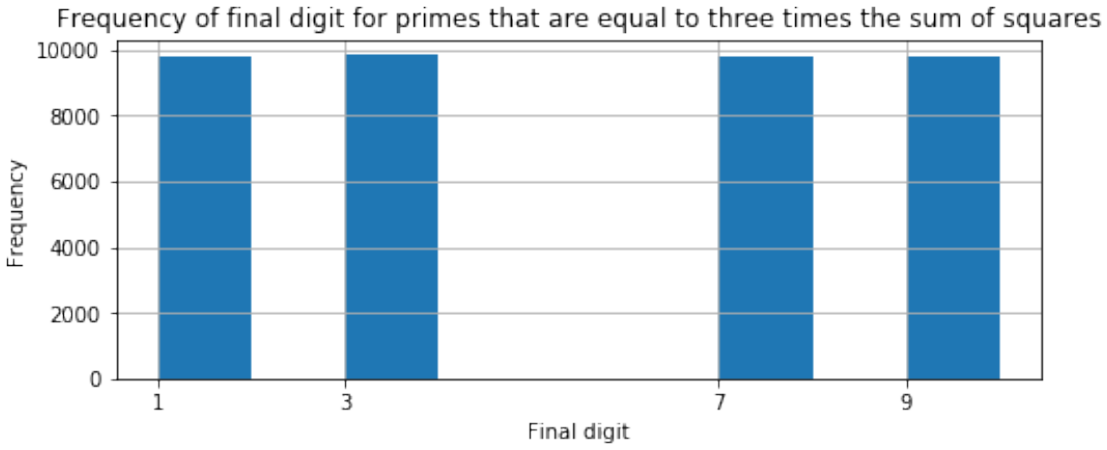



```
In [30]: %matplotlib inline
         plt.figure(figsize=(8, 3))  #  Makes the resulting figure 12in by 5in.
         plt.hist(lastDigitFreq_plot(sumOfSquares2Faster(1000000)), bins=range(1,11)) #  Makes
         plt.ylabel('Frequency')
         plt.xlabel('Final digit')
         plt.xticks([1,3,7,9])
         plt.grid(True)
         plt.title('Frequency of final digit for primes that are equal to two times the sum of
         plt.show()
```

```
Number of Primes =  39206
```




```
In [31]: %matplotlib inline
         plt.figure(figsize=(8, 3))  #  Makes the resulting figure 12in by 5in.
         plt.hist(lastDigitFreq_plot(sumOfSquares3Faster(1000000)), bins=range(1,11)) #  Makes
         plt.ylabel('Frequency')
         plt.xlabel('Final digit')
         plt.xticks([1,3,7,9])
         plt.grid(True)
         plt.title('Frequency of final digit for primes that are equal to three times the sum
         plt.show()

Number of Primes =  39232
```



We can therefore see that in every case there is a fairly even distribution between the four posibilities. In the first case we find that primes ending in one compose of 24.9%, ending in 3 compose of 25.1%, ending in seven compose of 25.1%, ending in nine compose of 24.9% of the

total number or primes. In fact for all three cases there seems to always be slightly less than 25% distribution for primes ending in one and nine, and slighty more than 25% distribution for primes ending in three and seven.

5 Conculsion

By first accomplishing the Ulam's spiral, we saw that there are diagonal lines that are values of a quadractic function like Euler's prime generating function. When one considers mod 4 of the primes and associates a color to that mod, we were able to interpret more information. The negative slope diagonal lines alternate between lines that contain primes of solely 1 mod 4 and 3 mod 4. This supported by the fact that the positive slope diagonal lines contains primes but in that same line the primes alternate mod 4.