

## 0.1 Introduction

This report details the approach and implementation details relating to the first coursework project of MATH36032. Please note that all programming and testing was carried out using GNU Octave rather than MATLAB.

## 0.2 The Euler-Mascheroni constant

### 0.2.1 Context

The Euler-Mascheroni constant is a mathematical constant that first appeared in Euler's writings in 1735. It is usually denoted  $\gamma$ , first appearing thus by Mascheroni in 1790 [1]. Formally, it is defined as follows.

$$\gamma = \lim_{n \rightarrow \infty} \left( -\ln n + \sum_{k=0}^n \frac{1}{k} \right) \quad (1)$$

Very little is known about this constant, it isn't even known whether or not it's irrational. Hence it is often convenient to approximate it's value to a quotient of integers.

Of course, since we do not know whether  $\gamma$  is even rational, while finding this quotient of integers, which we will denote  $\frac{p}{q}$  where  $p, q \in \mathbb{Z}$ , we must define a constraint on  $p$  and  $q$ . Since if  $\gamma$  is irrational, for each  $p, q$  we find, we'll be able to find another pair which is closer to  $\gamma$ .

Let's introduce an arbitrary integer  $N$ , we will aim to find a quotient of integers  $\frac{p}{q}$  such that  $p + q \leq N$ . Where  $\frac{p}{q}$  is the best approximation of  $\gamma$  with this constraint. Furthermore, where there are multiple pairs that give the best approximation, we will take the pair with the smallest value of  $p + q$ .

### 0.2.2 The brute force approach

The approach we will use to find the best integers  $p$  and  $q$  given the constraint, is to first look at a brute force method. Which is to try ever possible combination of  $p$  and  $q$  and see which one minimises  $|\gamma - \frac{p}{q}|$ . Then we will aim to optimise this approach.

Listing 1: A Brute force implementation to find p and q

```
1 function [p, q] = AppEmBruteForce(N)
2     emConstant = 0.577215664901533;
3
4     currentBestDiff = 1;
5     currentBestPQ = [0, 0];
6
7     % Try every valid combination of p and q
8     for q = 1:N
9         % Bound p by N - q so that p + q <= N
10        for p = 1:N-q
```

```

11
12         diffToEM = abs(p / q - emConstant);
13
14         if (diffToEM < currentBestDiff)
15             currentBestDiff = diffToEM;
16             currentBestPQ = [p, q];
17         end
18
19     end
20 end
21
22 p = currentBestPQ(1);
23 q = currentBestPQ(2);
24
25 end

```

The first thing we notice about the code in Listing 1 is it's inefficiency. In Big-O notation we refer to this as being  $O(N^2)$ . Since an increase in  $N$  results in an increase proportional to  $N^2$  in the number of operations performed.

### 0.2.3 Optimizing

We can do better than this. The aim here, is for a given  $q$ , to limit the values of  $p$  that we need to check.

First lets setup some notation. Notice that in Listing 1, when we find a better approximation, we update the *currentBestPQ* variable. Let's denote  $p_0$ ,  $q_0$  as the first values of this variable, and  $p_i$ ,  $q_i$  to be the  $i^{th}$  values of this variable. Hence the final value in this sequence is our answer. Let;

$$\gamma_i = \frac{p_i}{q_i} \quad (2)$$

Given a value of  $\gamma_i$ , we know that  $\gamma_{i+1}$  will be closer to the real  $\gamma$  than  $\gamma_i$ , since otherwise it wouldn't be one of the sequence. Hence;

$$|\gamma - \gamma_{i+1}| < |\gamma - \gamma_i|$$

Another way of saying this is as follows. Let  $\delta_i = |\gamma - \gamma_i|$ .

$$\gamma - \delta_i < \gamma_{i+1} < \gamma + \delta_i$$

Now if we substitute in equation 2 and multiply through by  $q_{i+1}$

$$q_{i+1}(\gamma - \delta_i) < p_{i+1} < q_{i+1}(\gamma + \delta_i) \quad (3)$$

Which is exactly what we need, for a given  $q$ , we have a constraint on  $p$ . So every time we find a better approximation, (a new  $\gamma_i$ ), we tighten the bound on which values of  $p$  we need to check.

However, to find  $\gamma_0$  currently we still have to check all possible values of  $p$  when  $q = 1$  since we don't yet have a current best approximation. Notice that if  $N = 1, 2$ , though valid inputs, the results aren't useful. However, for  $N \geq 3$ , observe that  $\gamma_0 = \frac{1}{2}$ . Hence we can just hard-code this initial

value. However, this does mean treating  $N = 1, 2$  as a special case, but which is worth it for the saved calculations.

*The reader may note at this point that it is often considered bad practice to hard-code values like this, as it sometimes affects flexibility. However, I argue that in this instance, we are hard-coding the value of a base case in our sequence of  $\gamma_i$  values, and that its performance improvement, combined with the legibility of the code makes it an acceptable approach.*

Given this justification, the code listing is as follows.

Listing 2: An optimized implementation to find p and q

```

1 function [p, q] = AppEm (N)
2     emConstant = 0.577215664901533;
3
4     % Check for the trivial case early to allow
5     % assumptions to be made further down
6     if (N <= 2)
7         p = N - 1;
8         q = 1;
9         return;
10    end
11
12    % Helper function for giving the difference between any
13    % value and the emConstant
14    absDiff = @(value) abs(value - emConstant);
15
16    % Since we've dealt with the trivial case of N = 1,2.
17    % The first value value of p/q will always be 1/2,
18    % we don't waste time calculating it but use it as the
19    % initial value
20    currentBestDiff = absDiff(1 / 2);
21    currentBestPQ = [1, 2];
22
23    % Start q off as 3 since we already accounted for q = 1, 2
24    q = 3;
25    qMax = N;
26
27    while (q < qMax)
28
29        % Limit the values of p that we check to only those that
30        % could be closer to the emConstant than our current best
31        pMax = min(floor(q * (emConstant + currentBestDiff)), N - q);
32        pMin = ceil(q * (emConstant - currentBestDiff));
33
34        for p = pMin:pMax
35
36            diffToEM = absDiff(p / q);
37
38            % Check if this approximation is better than the last one
39            if (diffToEM < currentBestDiff)
40                currentBestDiff = diffToEM;
41                currentBestPQ = [p, q];
42
43                % Recalculate the maximal value of q
44                qMax = ceil(N / (emConstant - currentBestDiff + 1));

```

```

45
46         end
47     end
48
49     q = q + 1;
50 end
51
52 p = currentBestPQ(1);
53 q = currentBestPQ(2);
54 end

```

There is one further optimization in Listing 2 that hasn't yet been discussed. This is the limiting of  $q$  as well as  $p$ . The reader may recall that in our brute force implementation (Listing 1), we allowed  $q$  to run from 1 to  $N$ , but it's clear that  $q = N$  (for non trivial values of  $N$ ) will never contribute to the best approximation.

By the definition of the problem and by Equation 3, we have the following information.

$$p + q \leq N$$

$$q(\gamma - \delta_i) < p$$

Hence combining these equation yields

$$q(\gamma - \delta_i) < p < N - q$$

$$\implies q(\gamma - \delta_i) < p \leq N - q$$

$$\implies q(\gamma - \delta_i + 1) < N$$

$$\implies q < \frac{N}{(\gamma - \delta_i + 1)}$$

So everytime we find a new approximation for  $\gamma$ , we can also update the maximal possible value of  $q$ . In practice this provides less of an optimization than the constraining of  $p$ , but for large  $N$ , does have a noticeable effect.

#### 0.2.4 Performance

The performance improvement in our optimization has proven to be more than acceptable. Recall how we classified the original algorithm (Listing 1) to be an  $O(N^2)$  algorithm. The following experiments show that the optimized version resembles an  $O(N)$  time complexity. I.e. as  $N$  gets larger, the time for the algorithm to terminates increases in a linear fashion.

Figure 1 shows this. It shows the time taken for both algorithms to complete, for  $N = 1 \rightarrow 300$ .

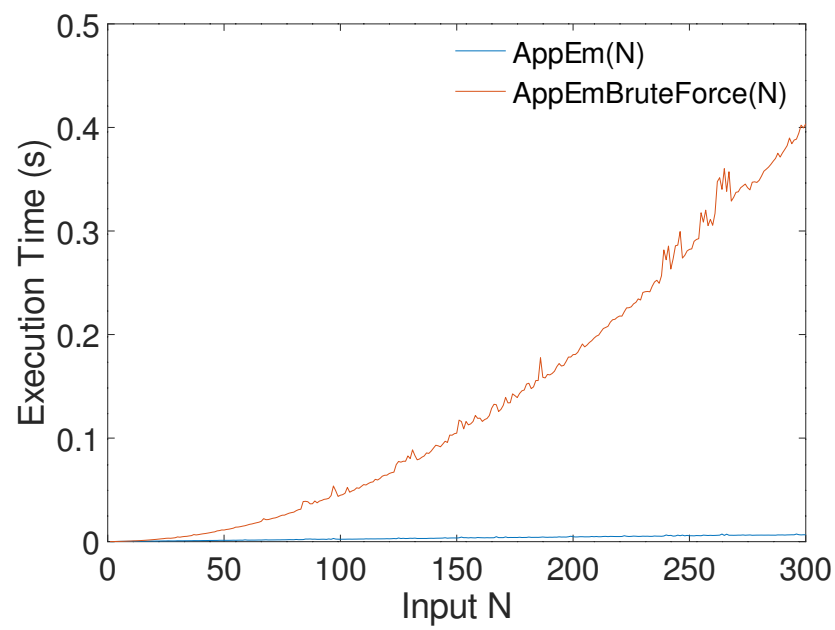


Figure 1: A performance comparison between the brute force algorithm and the optimized version

### 0.2.5 A specific example

Let's consider the case that  $N = 2019$ .

Listing 3: A test of both implementations for  $N = 2019$

```
1 tic();
2 [p, q] = AppEm(2019)
3 toc()
4
5 tic();
6 [p, q] = AppEmBruteForce(2019)
7 toc()
```

Running the code in Listing 3 gives the following output

```
>> test_2019
p = 228
q = 395
Elapsed time is 0.0487769 seconds.
p = 228
q = 395
Elapsed time is 18.8509 seconds.
```

We see the output from both approaches are in agreement, and that the optimizations result in an almost 400 times speed up.

## 0.3 Lucky numbers

Lucky numbers, (more commonly referred to as Carmichael numbers [2]) are integers  $n$  that satisfy the following criterion.

1.  $n$  is not prime
2. All prime factors of  $n$  are odd
3. All  $n$ 's prime factors are distinct i.e.

$$n = p_1 p_2 \dots p_k, \quad p_i \text{ is prime } \forall i, \text{ and } p_i \neq p_j \text{ for } i \neq j$$

4. For each prime factor  $p$  of  $n$ ,  $p - 1$  divides  $n - 1$

Our aim is to write a function to compute the smallest lucky number which is greater than or equal to an arbitrary integer  $N$ .

Firstly, some observations regarding the criterion.

- All prime factors of  $n$  are odd  $\iff n$  is odd  
(since the only even prime is 2, and if  $2|n$  then  $n$  is even).

Hence if  $n$  is even we can immediately rule out the possibility it is lucky.

- Given a list of the prime factors of  $n$ , if we find one prime factor  $p$  such that  $p - 1 \nmid n - 1$ , we can rule the number isn't a lucky number, and there is no need to check the rest of the prime factors.

Returning to our function, with these optimizations in mind, our approach is to check the integers greater than  $N$  until we find one that's lucky. We can see at once that the first number we must check is;

$$s = \begin{cases} N + 1, & \text{if } N \text{ is even} \\ N, & \text{if } N \text{ is odd} \end{cases}$$

Of course if this number is lucky, then we're done. But if not, what is the next number we must check? Notice that  $s + 1$  must be even, since  $s$  is odd by construction. We have already observed that even numbers can't be lucky. So the next number to check is  $s + 2$ . This pattern continues as we iterate, we can increment the number being checked by 2 each time, avoiding even numbers.

The implementation of the function is as follows.

Listing 4: The luckynum function

```

1 function n = luckynum(N)
2 % LUCKYNUM Find the smallest lucky number
3 % that is greater than or equal to N
4
5 % Ensure we start from the first odd number greater
6 % than or equal to N
7 % i.e N if N is odd, and N + 1 if even
8 current = N + rem(10, 2);
9
10 while(!isLucky(current))
11     current += 2;
12 end
13
14 n = current;
15 end

```

# Bibliography

- [1] Weisstein, Eric W. "Euler-Mascheroni Constant." From MathWorld—A Wolfram Web Resource. <http://mathworld.wolfram.com/Euler-MascheroniConstant.html> Referenced: 22-02-2019
- [2] Weisstein, Eric W. "Carmichael Number." From MathWorld—A Wolfram Web Resource. <http://mathworld.wolfram.com/CarmichaelNumber.html> Referenced: 23-02-2019