1 Introduction

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

2 The Euler-Mascheroni constant

2.1 Context

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

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

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 γ 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 γ is irrational, for each p, q we find, we'll be able to find another pair which is closer to γ .

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 γ with this constraint. Furthermore, where there are multiple pairs that give the best approximation, we will take the pair with the smallest value of $p+q$.

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

```
function [p, q] = AppEmBruteForce(N)
  emConstant = 0.577215664901533;

  currentBestDiff = 1;
  currentBestPQ = [0, 0];

  % Try every valid combination of p and q
  for q = 1:N
    % Bound p by N - q so that p + q <= N
    for p = 1:N-q

      diffToEM = abs(p / q - emConstant);

      if (diffToEM < currentBestDiff)
        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 inefficency. 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.

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;

$$\gamma_i = \frac{p_i}{q_i} \tag{2}$$

Given a value of γ_i , we know that γ_{i+1} will be closer to the real γ than γ_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}

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

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 γ_i), we tighten the bound on which values of p we need to check.

However, to find γ_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 **trivial values of N** ($N = 1, 2$) 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 offects flexiblilty. However, I argue that in this instance, we are hard-coding the value of a base case in our sequence of γ_i values, and that it's 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

```
function [p, q] = AppEm (N)
  emConstant = 0.577215664901533;

  % Check for the trivial case early to allow
  % assumptions to be made further down
  if (N <= 2)
    p = N - 1;
    q = 1;
    return;
  end

  % Helper function for giving the difference between any
  %value and the emConstant
  absDiff = @(value) abs(value - emConstant);

  % Since we've dealt with the trivial case of N = 1,2.
  % The first value value of p/q will always be 1/2,
  % we don't waste time calculating it but use it as the
  % initial value
  currentBestDiff = absDiff(1 / 2);
  currentBestPQ = [1, 2];

  % Start q off as 3 since we already accounted for q = 1, 2
  q = 3;
  qMax = N;

  while (q < qMax)

    % Limit the values of p that we check to only those that
    % could be closer to the emConstant than our current best
    pMax = floor(q * (emConstant + currentBestDiff));
    pMin = ceil(q * (emConstant - currentBestDiff));

    for p = pMin:min(pMax, N - q)

      diffToEM = absDiff(p / q);

      % Check if this approximation is better than the last one
      if (diffToEM < currentBestDiff)
        currentBestDiff = diffToEM;
        currentBestPQ = [p, q];

        % Recalculate the maximal value of q
        qMax = ceil(N / (emConstant - currentBestDiff + 1));

      end
    end

    q = q + 1;
  end

  p =  currentBestPQ(1);
  q =  currentBestPQ(2);
end
```

There is one further optimization in Listing 2 that hasn't yet been discussed. This is the constraining 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

$$\begin{aligned} & q(\gamma - \delta_i) < p < N - q \\ \implies & q(\gamma - \delta_i) < p \leq N - q \\ \implies & q(\gamma - \delta_i + 1) < N \\ \implies & q < \frac{N}{(\gamma - \delta_i + 1)} \end{aligned}$$

So everytime we find a new approximation for γ , 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.

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 $N = 1 \rightarrow 300$.

2.5 A specific example

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

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

```
tic();
[p, q] = AppEm(2019)
toc()

tic();
[p, q] = AppEmBruteForce(2019)
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.
```

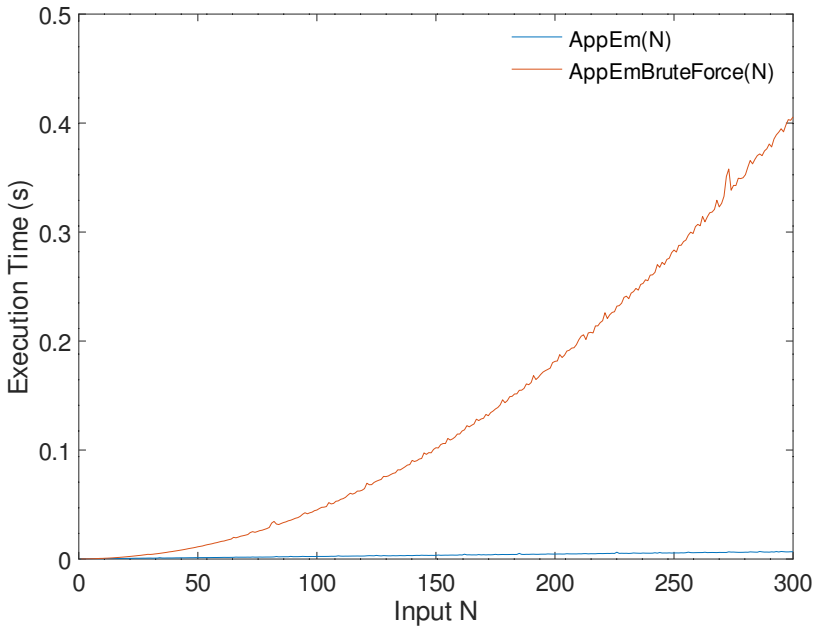

Figure 1: A performance comparison between the brute force algorithm and the optimized version

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

3 Lucky numbers

3.1 Context

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.

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

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.

3.2 The luckynum function

Returning to our function (which we'll call *luckynum*), with these observations 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 s we must check is;

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

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

```
function n = luckynum(N)
% LUCKYNUM Find the smallest lucky number
% that is greater than or equal to N

  % Ensure we start from the first odd number greater
  % than or equal to N
  % i.e N, if N is odd, and N + 1 if even
  current = N + 1 - rem(N, 2);

  while(!isLucky(current))
    current += 2;
  end

  n = current;
end
```

What remains, is to explore how we actually check if a given number n is lucky. In the luckynum function in Listing 4 we use the function *isLucky* which we will now explore.

3.3 The isLucky function

The full source is listed in Listing 5.

Listing 5: The isLucky function

```
function result = isLucky(N)

  result = true;

  % Return false if N is even
  if (mod(N, 2) == 0)
    result = false;
    return;
  end

  primeFactors = factor(N);
  factorsCount = length(primeFactors);

  % Check N has at least 2 prime factors and that
  % they're distinct
  if factorsCount == 1 ...
      || factorsCount != length(unique(primeFactors))

     result = false;
  else

     % Check all factors divide N - 1
     for factor = primeFactors
        if (mod(N - 1, factor - 1) != 0)
           result = false;
           break;
        end
     end
  end
end
```

The approach here is to initialise the result variable to true, and then as soon as one of the checks on N fails (i.e. determines N isn't lucky), result is set to false and crucially *no other checks are performed*.

First we check if N is even, the reader may wonder why this is needed. Since the implementation of the *luckynum* function never calls *isLucky* with even numbers, hence this check is redundant. However we aim to produce reusable code, and the next time we use the *isLucky* function, it may be called with an even number and so it is best practice to handle this correctly.

The order of the following checks is important, not in terms of correctness, but in terms of performance. N is first factorized and the result stored. To check if the number is prime, we must only check if the number of factors is 1, which is the fastest of the operations we must perform.

Here we take advantage of *lazy evaluation* to save making comparisons we don't need to. In a logical test

$$p||q$$

If p is evaluated to be true, then Octave knows that the whole expression must be true, and doesn't evaluate q. So, if we have more than one prime factor, we then check if the number of prime factors is equal to the number of unique prime factors, i.e. whether there are any duplicates.

The last, and clearly slowest check is the last of the criterion for lucky numbers. Notice that we make use of our observation that we can stop if we find any factors without the desired property in the code, in the form of a break in the loop.

The reader may notice that the most expensive operation in the implementation is the factorization of of N itself and whether there are any checks we could make before this. An alternative approach is to first check if the number is prime, and only if it is not do we do the factorization.

At first glance this seems a better solution since checking if a number is prime is faster than factorization. However, we must also consider that most numbers are not prime, so this check will fail more often than it passes, and it is in itself not a very fast operation. Hence the most common case is doing *both* the factorization, and the primality check.

The implementation of this alternative approach is listed in Appendix A, but more importantly Figure 2 shows a performance comparison between the two approaches. We can see clearly that our original approach is the more efficient in the general case.

3.4 A specific example

The following shows the output the *luckynum* function with the input of 2019.

```
>> luckynum(2019)
ans =  2465
```

Appendix A Alternative isLucky function

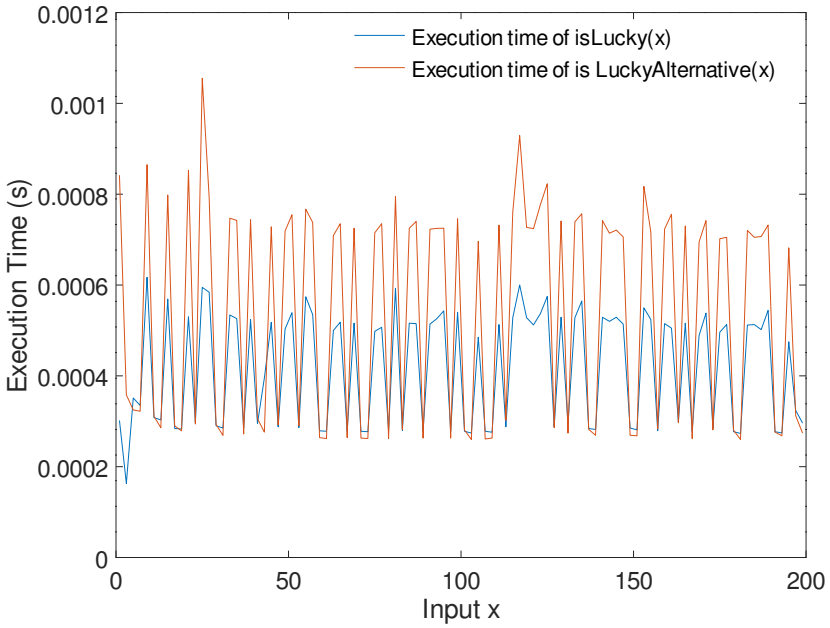

Figure 2: A performance comparison between isLucky and isLuckyAlternative. Note that the even values of x where skipped in producing the data in order to make it more legible, since both functions treat even numbers in the same way

Listing 6: The alternative implementation of isLucky

```
function result = isLuckyAlternative(N)

  result = true;

  % Return false if N is even or prime
  if (mod(N, 2) == 0 || isprime(N))
    result = false;
    return;
  end

  primeFactors = factor(N);
  factorsCount = length(primeFactors);

  % Check has distinct prime factors
  if factorsCount != length(unique(primeFactors))

     result = false;
  else

     % Check all factors divide N - 1
     for factor = primeFactors
        if (mod(N - 1, factor - 1) != 0)
           result = false;
           break;
        end
     end
  end
end
```

References

[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