PSBC Project 1

March 3, 2019

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.

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 in a paper by Mascheroni in 1790 [1]. Formally, it is defined as follows.

$$\gamma = \lim_{n\to\infty}\left(-\ln n + \sum_{k=0}^{n}\frac{1}{k}\right) \approx 0.577215664901533. \tag{1}$$

Very little is known about this constant, it isn't even known whether or not it is rational. Hence it is often convenient to approximate it's value to a quotient of positive integers, which we will denote $\frac{p}{q}$. However, if γ is indeed irrational, for each p, q we find, we'll be able to find another pair, who's ratio is closer to γ. The question becomes, which integers do we choose?

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 to solve this problem 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
    for p = 1:N

      if (p + q > N)
        continue;
      end

      diffToEM = abs(p / q - emConstant);

      if (diffToEM < currentBestDiff ||
          (diffToEM == currentBestDiff && p < currentBestPQ(1)))
        currentBestDiff = diffToEM;
        currentBestPQ = [p, q];
      end

    end
  end

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

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 the number of operations performed is proportional to N^2.

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 we will set up 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

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

Lets assume we already have a γ_i. There are two possibilities. Either, γ_{i+1} is an equally good approximation, but $p_{i+1} + q_{i+1} < p_i + q_i$. Or γ_{i+1} is closer to γ than γ_i.

Case 1

Formally we have the following two equations.

$$|\gamma - \gamma_{i+1}| = |\gamma - \gamma_i|, \tag{3}$$

$$p_{i+1} + q_{i+1} < p_i + q_i. \tag{4}$$

From here, we need to make the assumption that γ is in fact irrational. The justification for this is that if γ is rational, then we can define it exactly as a quotient of integers, making the whole exercise redundant.

In this case we can see that if Equation 3 holds, it must be the case that both γ_i and γ_{i+1} are less than γ, or they're both greater than γ. Since if one was greater than γ and one less, then equation 3 would contradict γ being irrational.

This directly implies that

$$\gamma_{i+1} = \gamma_i \implies \frac{p_i}{q_i} = \frac{p_{i+1}}{q_{i+1}}.$$

However, the variable `q` in the code is strictly increasing. Hence the sequence of q_i is strictly increasing. So $q_i < q_{i+1}$. It then follows directly that $p_i < p_{i+1}$ and hence $p_i + q_i < p_{i+1} + q_{i+1}$. This contradicts Equation 4.

Hence we can see that this case will never actually occur in practice. This means we can define the next term in our sequence as in *case 2*.

Case 2

Given the argument in *case 1*. We know that γ_{i+1} will be strictly 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 $\delta_i = |\gamma - \gamma_i|$, then

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

Now if we substitute Equation 2 and multiply through by q_{i+1}, the result is

$$q_{i+1}(\gamma - \delta_i) < p_{i+1} < q_{i+1}(\gamma + \delta_i). \tag{5}$$

What we have now, is for a given q we have a bound on the possible values of p we need to check. Furthermore, we have a way of tightening the bound every time we find a new approximation for γ.

Observe that to find γ_1, it seems 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 $N = 1, 2$ are valid inputs, but the results are

$$\frac{p}{q} = \frac{0}{1} \text{ and } \frac{p}{q} = \frac{1}{1} \text{ respectively.}$$

Which are not useful approximations of γ. However, for $N \geq 3$, observe that $\gamma_1 = \frac{1}{2}$. Hence we can just hard-code this initial value. This does mean treating **trivial values of N** ($N = 1, 2$) as a special case.

The reader may note at this point that it is often considered bad practice to hard-code values like this, as it sometimes effects 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.

Finally, there is another optimization we can perform. This is the constraining of q as well as p. The reader may recall that in our brute force implementation (Listing 1), we allowed the variable `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, suggesting we should be able to define an upper bound on q which is less than N.

Indeed, by returning to Equation 5. If we look at the left hand side of this constraint;

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

and recall our original constraint for each p_j and q_j,

$$p_j + q_j \leq N.$$

Combining these equations yields

$$\begin{aligned} q_{i+1}(\gamma - \delta_i) < p_{i+1} \leq N - q_{i+1} \\ \implies q_{i+1}(\gamma - \delta_i) \leq N - q_{i+1} \\ \implies q_{i+1}(\gamma - \delta_i + 1) \leq N \\ \implies q_{i+1} \leq \frac{N}{(\gamma - \delta_i + 1)} \end{aligned}$$

So everytime we find a new approximation for γ, we can also update the maximal possible value of q (`qMax` in Listing 2). In practice this provides less of an optimization than the constraining of p, but for large N this does have a noticeable effect.

Given this justification the code is included in Listing 2.

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
```

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.

Figure 1 shows the time taken for both algorithms to complete for the input N from 1 to 250. It appears to support our claim, but to further confirm the validity of our analysis, Figure 2 shows the logs of the values in Figure 1 on logarithmic axes. The gradients of the lines approximating the brute force curve and optimized curve are **1.9** and **0.9**, respectively. This is almost exactly what we'd expect from taking the logs of quadratic and linear plots.

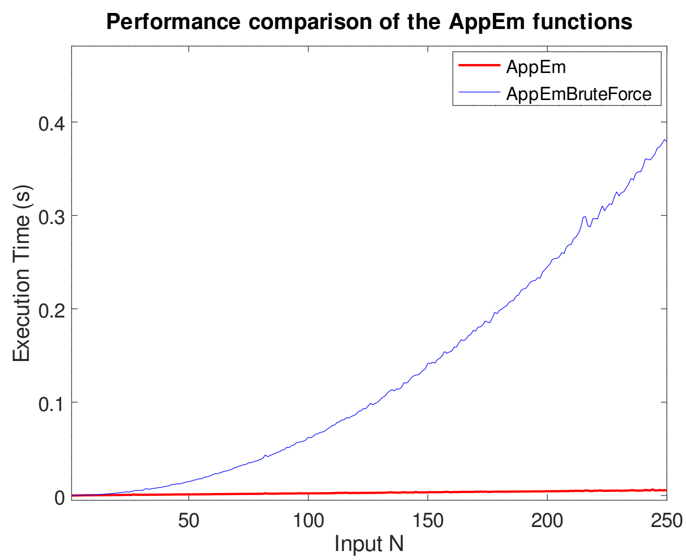

Figure 1: A performance comparison between the brute force algorithm and the optimized version.

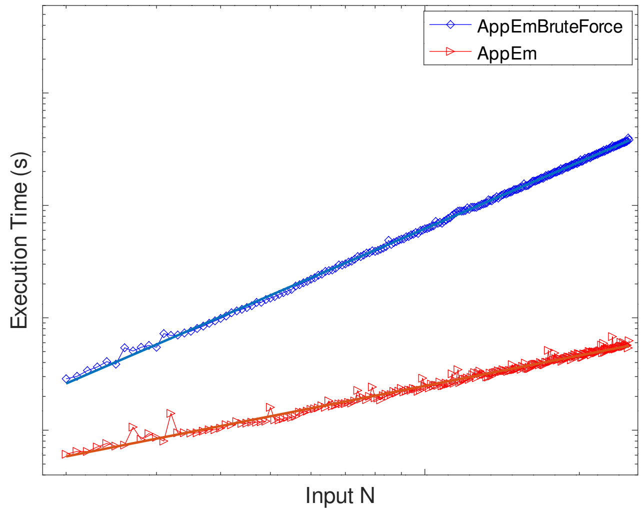

Figure 2: A graph of the logs of the curves in figure 1, plotted on logarithmic axes. The axis scales are omitted since the important feature here is the gradient of the lines.

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.
```

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. The integer n is not prime.
2. All the prime factors of n are odd.
3. All n's prime factors are distinct.
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 positive integer N.

Firstly, some observations regarding the criterion.

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

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

- If we find one prime factor p of n such that $p-1$ **does not** divide $n-1$, we can rule that n isn't a lucky number. Furthermore, there is no need to check the rest of the prime factors.

3.2 The `luckynum` function

With these observations in mind, we now aim to write a function (which we'll call **`luckynum`**) to achieve our goal. The general approach we will take 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

```
% LUCKYNUM Find the smallest lucky number
% that is greater than or equal to N
function n = luckynum(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 determine how we check if a given number 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)

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

  result = true;
  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
```

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 if we make use of the `isLucky` function in future, 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. 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 the following logical test with arbitrary logical conditions p and q;

```
if p || q
...
```

If p is evaluated to be true, then Octave knows that the whole expression must be true, and doesn't evaluate q.

In this case, if our check, `factorsCount == 1` evaluates to false, only then do we check if the prime factors are unique. If the `factorsCount` was 1, the `if` statement wouldn't evaluate the second condition.

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.

The reader may notice that the most expensive operation in the implementation is the factorization of `N` itself. Leading us to question if there are any more checks that can be performed before factorization. An alternative approach is to first check if the number is prime, and only factorize it if not.

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 3 shows a performance comparison between the two approaches. We can see clearly that our original approach is generally faster, especially for larger inputs.

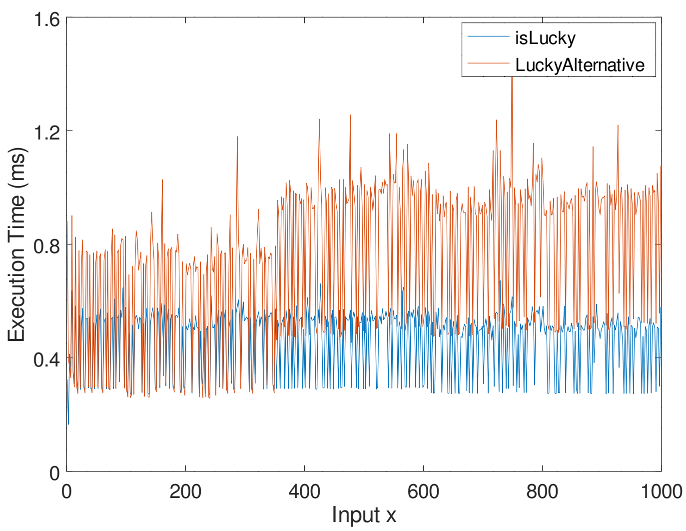

Figure 3: A performance comparison between `isLucky` and `isLuckyAlternative`.

3.4 A specific example

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

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

4 Perfect Numbers

4.1 Context

Given a positive integer n we say that n is *perfect* if it's square consists of all the digits 1 to 9 exactly once. For example, when $n = 24237$,

$$n^2 = 587432169.$$

So 24237 is said to be a perfect number.

The problem we're interested in solving is as follows, given an arbitrary positive integer N with the following bounds,

$$10^8 < N < 10^9. \tag{6}$$

We seek the perfect number n who's square is closest to N.

4.2 The `isPerfect` function

Firstly, we need a way of checking whether a given integer n is perfect. The function is listed below (Listing 6).

Listing 6: The `isPerfect` function

```
% Checks whether the square of a given number contains
% all nine digits exactly once
function isPerfect = isPerfect(value)
  digitsArr = int2Digits(value ^2);

  isPerfect = numel(digitsArr) == 9 ...
                 && !any(digitsArr == 0) ...
                 && numel(unique(digitsArr)) == 9;
end
```

Once again we make use of the lazy evaluation property of the language. We see that three checks are made on the digits.

- Checking there are 9 digits exactly.
- Checking that the digit 0 doesn't appear.
- Checking each digit appears exactly once.

The checks are done in order of speed, this way, because of lazy evaluation, more expensive checks won't be carried out if cheaper checks rule out the possibility of `value` being perfect.

We have neglected so far to comment on the `int2Digits` function. Which returns an array of the digits of a base 10 number. (The full listing is included in Appendix 9).

The obvious alternative here would be to use the inbuilt `num2str` function in a similar way. However, having performed some experiments comparing the two approaches, the custom approach appears to be marginally faster.

4.3 The `PerfNum` function

Now that we have a mechanism for determining if a given number is perfect, we can return to the original problem.

Let $n_1, n_2, ..., n_k$ be the perfect numbers in ascending order without repetitions. Our approach will be to develop an algorithm under the assumption that N lies between the squares of two perfect numbers. So suppose for some $i \in \{1, .., k-1\}$

$${n_i}^2 \leq N \leq {n_{i+1}}^2.$$

Then we will observe that our algorithm also returns the correct result if $N < {n_1}^2$ or $N > {n_k}^2$.

Clearly, if $i < j$ then $n_i < n_j$, so

$${n_1}^2 < {n_2}^2 < ... < {n_i}^2 \leq N \leq {n_{i+1}}^2 < ... < {n_k}^2.$$

Hence by taking square roots we obtain

$$n_1 < n_2 < ... < n_i \leq \lfloor \sqrt{N} \rfloor \leq n_{i+1} < ... < n_k. \tag{7}$$

Recall that we defined n to be the perfect number who's square is closest to N. Clearly, by Equation 7, we know that either $n = n_i$ or $n = n_{i+1}$. Let $A_j = \{\lfloor \sqrt{N} \rfloor + j, \lfloor \sqrt{N} \rfloor - j\}$.

Now we iterate through $A_0, A_1, A_2, ...$ Observe that our target n, is an element of A_q where A_q is the first set we come across (i.e. minimal q) that contains a perfect number. If both elements are perfect, we take n to be the one who's square is closest to N.

Now notice that if originally $N < {n_1}^2$, then the correct answer is $n = n_1$. By iterating over the sets, it's clear that the first set A_j that contains a perfect number, contains n_1. Similarly if $N > {n_k}^2$. Hence our algorithm is guaranteed to find the correct value of n.

The code listing is below (Listing 7). In the code the sets aren't explicitly defined, but the loop structure does iterate over their elements.

Listing 7: The `PerfNum` function

```
% PERFNUM returns n, such that n^2 is a perfect square
% number with all nine distinct digits from 1 to 9,
% which is closest to the input N, another nine digits number
%(can be assumed to be between 10^8 and 10^9 -1)
function n = PerfNum (N)
  rootN = round(sqrt(N));

  currentOffset = 0;
  found = false;

  while (!found)

    perfects = [];

    for current = [rootN + currentOffset, rootN - currentOffset]
        if (isPerfect(current))
          perfects = [perfects, current];
        end
    end

```

```
    % Find the index of the perfect number
    % who's square is closest to N
    [delta, indexOfClosest] = min(abs(N - perfects .^ 2));

    if (indexOfClosest > 0)
      n = current;
      found = true;
    end

    currentOffset = currentOffset + 1;
  end
end
```

For this function the assumption in Equation 6 is made. Hence the worst case performance of this function is when `N` is exactly between the squares of the perfect numbers n_i and n_{i+1} which have the most distance between them. We can fairly easily work out that these are 20316 and 22887.

Let's try the worst case value of `N`, which we can work out from this is `N = 468277312`.

```
tic();
PerfNum(468277312)
toc()
```

```
 >> testWorstCase
ans =  22887
Elapsed time is 0.859387 seconds.
```

It's clear that the function in the very worst case still runs in under a second, hence for any valid input it will return in an acceptable amount of time.

4.4 Another specific example

The following shows the output of the `PerfNum` function with the input 360322019.

```
>> PerfNum(360322019)
ans =  19023
```

Appendix A Alternative implementation of `isLucky`

Listing 8: 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
```

Appendix B The int2Digits function

Listing 9: The int2Digits function

```
% Given a base 10 integer N, returns
% an array of it's digits
function digitsVector = int2Digits(N)

  digitsVector = [];
  while (N != 0)
    digitsVector = [mod(N, 10), digitsVector];
    N = floor(N / 10);
  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