Number Theory in MATLAB

Yann McLatchie
Student ID: 10161640

Introduction

This report will treat the three problems given in the first project chronologically, at each step reformulating the problem and providing the mathematical background necessary to tackle it, before presenting the problem solving approach. The code and its result will then be presented. Once we have solved all three problems, we will critically analyse each of the solutions and discuss potential improvements.

1 The Euler-Mascheroni Constant

Background and Problem Formulation

The first problem on the sheet deals with the Euler-Mascheroni constant, defined as

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

We are tasked with finding the pair of positive integers $p, q \in \mathbb{Z}^+$ such that $|{}^p/_q - \gamma|$ is the smallest possible given that $p + q$ is less that or equal to our input N, a positive integer greater than 1. This is equivalent to finding the best rational approximation to the Euler-Mascheroni constant with the sum of the denominator and numerator constrained. In the case that two different pairs of rational numbers produce the same, best approximation, we choose the pair with the lower sum of numerator and denominator.

Problem Solving Approach

Given our input $N \in \mathbb{Z}^+$ we have three variables we would like to keep track of, namely the values of p and q that produce the best approximation, the accuracy of that approximation $|{}^p/_q - \gamma|$, and the sum $p + q$, which we label `bestPair`, `bestApprox`, and `bestPairSum` respectively. The values of `bestApprox` and `bestPairSum` we initiate to infinity so that we can later minimise them, and we initialise `bestPair = [NaN NaN]`, a null array to be modified upon finding a better pair.

We are able to solve this problem using an efficient search of the parameter space. We first cycle through all possible values of $p = 1, ..., N$ and for each p we cycle through all q such that $p + q \leq N$, equivalently $q = 1, ..., N - p$. This can be achieved with a nested `for` loop structure. Note that we do not need to loop through all N integers twice, thus improving run time.

We then check if for these values of p and q, our approximation is better than our current best with an `if` statement testing `abs(p/q - emconst) < bestApprox`. In the first iteration, this is necessarily the case since we initialised the error of our approximation to be infinite, so any finite error will be an improvement. In later iterations if we improve our approximation, we update our variables, and move on to the next pair of integers in the loops. In the case that we have not improved our current best approximation but we have achieved an approximation with equal accuracy and a lower sum $p+q$, which is tested with an `elseif` statement verifying `abs(p/q - emconst) == minNum` and `p+q < minPairSum`, we update the `bestPairSum` and `bestPair` variables. Finally, if we do not improve on our past approximation on a given iteration, that is if neither of the two previous `if-elseif` statements are satisfied, we make no modifications to our variables and move on to the next pair of integers. This logic repeats for the each step in the loops and once they are all completed, we return the final best approximation.

This algorithm has quasilinear time complexity, which was verified by timing the function with large N. A less efficient algorithm would loop through all values 1 to N for both p and q values with an added `if` statement to verify `p+q <= N`, which would have quadratic time complexity $O(N^2)$. Knowing that we verify each pair of integers p and q that satisfy $p+q \leq N$ using the limits in our for loops, and having dealt with all possible cases in our `if-elseif` statements, we are confident that the code returns the correct result.

Code and Results

Implementing the logic described above, we arrive at the following MATLAB code in Figure 1 which has been commented for readability. Running this, we find that we get the correct answer for the example case, and achieve the following for the test case:

```
>> AppEm(2021)
ans =
   228   395
```

2 MyLucky Number

Background and Problem Formulation

Second on the sheet is the function `MyLuckynum`. In this problem, we are asked to find the smallest MyLucky number greater than or equal to our input N. These MyLucky numbers are known by a different name in Number Theory, namely they are the Carmichael numbers, defined as follows in Riesel (1994).

Definition 2.1 (Carmichael Numbers) *A Carmichael number is a composite number n such that for all integers b relatively prime to n*

$$b^{n-1} \equiv 1 \mod n.$$

```
function bestPair = AppEm ( N )
    % Approximate the Euler-Mascheroni constant by the rational
    % number p/q, among all pairs of positive integers (p, q) such
    % that p+q <= N, N a positive integer greater than 1.

    emconst = double(eulergamma);
    bestApprox = Inf;
    bestPairSum = Inf;
    bestPair = [NaN, NaN];
    for p = 1:N
        % p+q <= N implies q <= N-p
        for q = 1:N-p
            % if we have found a more accurate approximation
            if abs(p/q - emconst) < bestApprox
                bestApprox = abs(p/q - emconst);
                bestPairSum = p + q;
                bestPair = [p, q];
            % if we have an equivalent approximation with a lower sum
            elseif (abs(p/q - emconst) == bestApprox) && (p+q < bestPairSum)
                bestPairSum = p + q;
                bestPair = [p, q];
            end
        end
    end
end
```

Figure 1: Code listing for the `AppEm` function.

We are given a different formulation of this definition in the project according to Korselt (1899).

Definition 2.2 (Korselt's Criterion) *A positive composite integer n is a Carmichael number if and only if n is square-free, and for all prime divisors p of n, $p-1 \mid n-1$.*

From which we can see that a number n is Carmichael if and only if it has distinct prime factors, all of which are odd, and each of these prime factors satisfy the divisibility criterion given above. We will use this simpler criterion to identify Carmichael numbers and return the next one larger than our input.

Problem Solving Approach

The algorithm we will use to solve this problem is very simple: Given our input N we first check if it is a Carmichael number. If it isn't then we increment it by 1 and test again, repeating this process until we find a Carmichael number, which we then return as our output. The difficulty in the problem is then efficiently determining whether or not a number is indeed Carmichael.

Using Korselt's criterion given in the introduction, we are able to find these numbers more efficiently than the definition in Riesel (1994). In order to do so, we first treat the trivial edge case of $N = 1$, which is clearly not a Carmichael number since it is noncomposite. For N larger than 1, we have that N is a Carmichael number if and only if it has distinct prime factors where each prime factor is odd, and further that for each prime factor p, $p-1 \mid N-1$.

In order to verify these conditions, we implement the native `factor` function in MATLAB to efficiently retrieve the prime factors of N. We then verify that they are distinct, odd, and all satisfy the divisibility criterion. To test that they are all distinct, we use the native `unique` function which, given a some data array `A`, returns it without repetitions. By comparing the length of the array of unique values with the original data using the native `length` function, we can see how many values are repeated. Specifically, if they are of the same length, the original data array consists only of unique values. We can test if all the prime factors are odd using vector logic and the `all` function. First building a boolean vector which contains, in the i^{th} index, either 0 if the i^{th} prime factor is even, and 1 if it is odd, the `all` function then tests if all elements in this vector are true or non-zero. We test if an element is odd by testing if its modulus base 2 is different to 0, in other words if it is divisible by 2. Finally, we employ a similar methodology to test if each prime factor satisfies $p - 1 \mid n - 1$, only using a different modulo base. If a number passes these tests, we conclude that it is indeed a Carmichael number. Note that we only test the divisibility $p - 1 \mid n - 1$ after testing that our input is not prime and has distinct and odd prime factors so as to speed up run time.

Code and Results

Implementing this logic in MATLAB, we achieve the function shown in Figure 2. Running this code for the test case, we achieve the following result.

```
>> MyLuckynum(2021)
ans =
    2465
```

Having presented the logic behind the code, we can be confident that it will produce the correct output. This final output was checked against Sloane in the Online Encyclopedia of Integer Sequences (Sequence A002997) and identified the correct Carmichael number.

3 Beautiful Squared Number

Background and Problem Formulation

The third problem is to do with Beautiful Squared Numbers, once more a pseudonym of a relevant series of numbers, namely the zeroless pandigital numbers: numbers who, when squared, contain the digits 1-9 exactly once and no 0's. As with question 2, we can cross-check our results against Weisstein and Beedassy in the OEIS (Sequences A050289 and A071519 respectively).

We are asked to, given a positive integer N, return the number n for whom n^2 is the zeroless pandigital number closest to N. Note that not all zeroless pandigital are squares, so we are in fact operating on a subset of the series A050289.

Problem Solving Approach

The main algorithm we will use to solve this problem is as follows: We take the rounded value of the square root of out input N, using the `round` and `sqrt` functions, as the center

```
function n = MyLuckynum ( N )
    % Find the lowest Carmichael number greater than or equal to N
    % https://oeis.org/A002997

    function result = oddAndDistinctPrimeFactors ( n )
        % verify that the prime factors of n are all odd and distinct

        primeFactors = factor(n);
        if (length(primeFactors) == length(unique(primeFactors))) && ...
                (all(mod(primeFactors,2) == 1))
            result = 1;
        else
            result = 0;
        end
    end

    function result = isCarmichael ( n )
        % check whether n is a Carmichael number using Korselt's criterion

        result = 0;
        % deal with trivial case
        if n == 1
            return
        % test Korselt's criterion
        elseif (oddAndDistinctPrimeFactors(n)) && (~isprime(n))
            p = factor(n);
            if all(mod(n-1, p-1) == 0)
                result = 1;
                return
            end
        end
    end

    % main while loop
    n = N;
    while ~isCarmichael(n)
        n = n + 1;
    end
    return
end
```

Figure 2: Code listing for the `MyLuckynum` function.

of our search space, denoted `n`. We then introduce two new search variables `a` and `b` which we initialise to be equal to `n`, and enter a `while` loop. In this loop, we check if either `a^2` or `b^2` are zeroless pandigital numbers and while this is not the case we increment them away from the search centre `n` by 1 in opposite directions. In other words, at the end of every iteration in the while loop, `a = a + 1` and `b = b - 1`. When at least one of them is indeed a zeroless pandigital number, we return it. If both `a` and `b` satisfy the criterion (for example in the case $N = 0$, which is beyond the constraints for the project) we return `a`, the positive integer such that its square is a zeroless pandigital number. By searching outwards from a central point, which we take as a function of our input, we search the space in the most efficient way possible.

In order to verify whether or not a given number is zeroless pandigital, we need check that it contains 9 digits, that each of these digits is distinct, and that none of these digits is 0 using a single `if` statement. We are able to separate an input integer into its digits with

the user-defined `num2dig` function, and from this check these three criteria. The `num2dig` function takes an input integer n, and initialises an empty array `dig`. We iteratively append the remainder after dividing 10 into n to the beginning of `dig`, and modify n to be the integer part of this division at each step while n is greater than 0, thus building an array of the digits of n. Once we have separated our integer into an array of its digits, we can once more use the `length` function to test that its length is 9, and that it consists only of unique digits using a similar methodology to that in problem 2 with the `unique` function. Testing also that 0 is not one of these digits with the `ismember` function, we can determine whether our input does indeed include all digits 1 to 9 exactly once. As such, testing whether a number is a zeroless pandigital number is very quick. Our algorithm is efficient overall due to the radial nature of our search of the parameter space and the ease with which we verify if a number is zeroless pandigital.

Code and Results

This logic is translated into MATLAB as follows in Figure 3. Running this code for the test case, we achieve the following result.

```
>> Beautisqnum(360322021)
ans =
       19023
```

We have confidence in this result, having compared the outputs of our `isBeautiful` function against Weisstein, and since we are able to compare this final result against Beedassy in the OEIS.

Critical Analysis

Problem 1

The largest flaw with the algorithm given the solve the first problem of approximating the Euler-Mascheroni constant, is the time complexity. While it is not quite quadratic with respect to N, it is still worse than linear given the need for two `for` loops. The search indices help restrict which numbers need to be searched, since we only cycle through all N numbers for p and then only $N-p$ for q at each of the N steps, but it nonetheless increases run time. This algorithm still computes an approximation faster than that presented in Brent and McMillan (2000) which is in $O(n(\log n)^3)$, thanks to the constraints of our function. Our algorithm does not require large memory allocations, since the only variables used are initiated before the loops. There is no obvious way to bypass the need for two `for` loops given the requirement of the problem, and so we are content with the efficiency of our solution.

Problem 2

In the `MyLuckynum` function, there is a nested function named `isCarmichael` which tests whether or not a given input is a Carmichael number. In order to do so, it must calculate the

```
function n = Beautisqnum ( N )
    % Find n whose square is the zeroless pandigital number closest to N
    % https://oeis.org/A071519

    function dig = num2dig( n )
        % convert an integer into a vector of its digits

        dig = [];
        while (n > 0)
            % append the remainder after dividing 10 into n to the
            % beginning of the vector, and modify n to be the division
            % quotient
            dig = [rem(n, 10) dig];
            n = floor(n/10);
        end
    end

    function result = isBeautiful( n )
        % test whether n is a zeroless pandigital number

        digs = num2dig(n);
        % test if n contains 9 unique digits, none of which is 0
        if (length(unique(digs)) == 9) && ...
                (length(unique(digs)) == length(digs)) && ...
                (~ismember(0, digs))
            result = 1;
        else
            result = 0;
        end
    end

    % set search center
    n = round(sqrt(N));
    a = n;
    b = n;
    % radial search
    while (~isBeautiful(a^2)) && (~isBeautiful(b^2))
        a = a+1;
        b = b-1;
    end
    % determine which is the beautiful square number
    if isBeautiful(a^2)
        n = a;
        return
    elseif isBeautiful(b^2)
        n = b;
        return
    end
end
```

Figure 3: Code listing for the `Beautisqnum` function.

prime factorisation of the input, and loop through each prime factor. Carmichael numbers can have many prime factors, and as the input increases the number of prime factors we have to check will also grow, increasing run time for very large numbers. In order to retrieve these prime factors, the `factor` function was used, which uses a sieve approach according to MATLAB (2021). If the input is large, it may require a large memory allocation although the algorithm is time efficient. The `while` loop in the main body of the code may cause issues as the input grows, since the distance between Carmichael numbers increases as the numbers grow larger, as can be seen in Sloane, so we have a larger area to search. Unfortunately

given the requirements of the question this loop is necessary since we are looking for the least Carmichael number greater than or equal to our input. We could make our algorithm run faster in some cases by replacing the `while` loop with a `for` loop that searches the next n numbers after our input for a Carmichael number. The advantage of this method is that we can assert a radius in which we must find a Carmichael number or return an error message, saving us from a potentially longer wait. Of course this would no longer answer the question posed, and in testing larger numbers there was never a long enough run time for me to seriously consider making such a modification to the code.

Problem 3

The `Beautisqnum` function has the same issue surrounding the `while` loop. That is that we could once more convert this into a `for` loop at the expense of accuracy, since in some cases we may not find a zeroless pandigital number in the search radius we set. The algorithm we use for finding a zeroless pandigital number runs in constant time, and thus poses few complexity issues.

We could have used the native `int2str` instead of `num2dig` to test the zeroless pandigital criteria, since both arrays and strings are compatible with the `length` and `ismember` functions. The choice to use `num2dig` was motivated by the readability of `ismember(0, digs)` as opposed to `ismember('0', digs)` with '0' a string for `int2str`. To avoid confusion about what data types we are working with, `num2dig` was chosen.

References

Lekraj Beedassy. Numbers whose square is a zeroless pandigital number. OEIS, `https://oeis.org/A071519`.

Richard P. Brent and Edwin M. McMillan. *Some New Algorithms for High-Precision Computation of Euler's Constant*, pages 448–455. Springer New York, New York, NY, 2000. ISBN 978-1-4757-3240-5. doi: 10.1007/978-1-4757-3240-5_50. URL `https://doi.org/10.1007/978-1-4757-3240-5_50`.

A. R. Korselt. *Problème Chinois.* 6 edition, 1899.

MATLAB. *Version 9.9.0 (R2020b) Update 4.* The MathWorks Inc., Natick, Massachusetts, 2021.

Hans Riesel. *Prime numbers and computer methods for factorization. 2nd ed*, volume 126. Boston, MA: Birkhäuser, 2nd ed. edition, 1994. ISBN 0-8176-3743-5/hbk.

N. J. A. Sloane. Carmichael numbers. OEIS, `https://oeis.org/A002997`.

Eric W. Weisstein. Zeroless pandigital numbers. OEIS, `https://oeis.org/A050289`.