MATLAB – Project 1

ID : 10203166
March 11, 2021

Here are the answers to questions 1-3, using the functions explained below.

```
[p,q] = AppEm(2021)
```

```
p = 228
q = 395
```

```
MyLuckynum(2021)
```

```
ans = 2465
```

```
Beautisqnum(360322021)
```

```
ans = 19023
```

Please also find attached the .m files to these functions, as well as numtodig(). The following are some test cases for the function Beautisqnum(). Due to space, the testcases to AppEm() and MyLuckynum() are not shown.

```
t1 = Beautisqnum(12312);
t2 = Beautisqnum(10^8);
t3 = Beautisqnum(10^8-1);
t4 = Beautisqnum(10^9);
t5 = Beautisqnum(0);
t6 = Beautisqnum(1);
t7 = Beautisqnum(-1231);
t8 = Beautisqnum(3/4*(10^8+10^9));
t9 = Beautisqnum((10^8+10^9)/2);
t10 = Beautisqnum((10^8+10^9)/3);
t11 = Beautisqnum((10^8+10^9)/5);
t12 = Beautisqnum((10^8+10^9)/4);
t13 = Beautisqnum((10^8+10^9)/9);

testcases = [t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12 t13]
```

```
testcases = 1×13
       11826       11826       11826       30384       11826       11826       11826       29034       23439       19023       14676       15963       11826
```

1 AppEm.

The AppEm function finds the smallest integer values p, q such that the absolute value of p/q is the best approximation for the Euler-Mascheroni constant, such that $p + q \leq N$. This algorithm is best understood by looking at the demonstration, using N=10.

Approach

1) Declare the variables P, emconst and bestapprox.
2) Work out a base case for the computation $\left|\frac{p}{q} - \gamma\right|$,.
    - Set the outputs p = 1 and q = 2.
    - Divides each element of P in the range 1 up to N-P with the use of the ./ operator.
    - It is up to N-P due to $p + q \leq N$.
3) Enter a while loop on the condition that P less than N-1 and the length of the array is greater than 2 (to not loop over empty arrays)
4) Compute abs(P./(P+1:N-P)-γ) and store it in arr. The range is now from P+1 as the base already checked the case when p=q.
5) Find the smallest value in arr, using the min() function. Store it in a variable named smallest.
6) If the smallest value is less than bestapprox then update bestapprox. Take note of the positions p and q.
    - p is equal to P.
    - To work out q, find the position in the arr and add P to account for the the values taken away from the beginning of arr. Due to (P+1:N-P).
7) If the smallest is not greater than bestapprox then increment P.

Demonstration.

Set a counter variable equal to 1, call it P, which will be incremented on each iteration in a while loop. Store the Euler-Mascheroni constant in a variable named emconst. Set *bestapprox* to intmax, MATLAB's largest value of the 32-bit signed integer type[1].

```
N =10; % this is for demonstration.
P = 1;
emconst = 0.577215664901533;
bestapprox = intmax;
```

Next, we enter a while loop. On the condition that P is always less than or equal to N-1. The crux of this algorithm works on MATLAB's ./ operator[2]. Compute $\left|\frac{p}{q} - \gamma\right|$, using the abs() function.

```
while P <= N-1
```

[1] The MathWorks, Inc., intmax, {1994-2021}, <https://uk.mathworks.com/help/matlab/ref/intmax.html>

[2] The MathWorks, Inc., intmax, {Retrieved February 25, 2021},, <https://uk.mathworks.com/help/matlab/ref/rdivide.html>

```
    arr = abs(P./(1:N-P) - emconst) % from P+1, as already checked the case
when p = q in base case, to N-P due to Q+P <= N
    smallest = min(arr);
    P = P+1;
 end
```

```
arr = 1×9
    0.4228    0.0772    0.2439    0.3272    0.3772    0.4105    0.4344    0.4522    0.4661

arr = 1×8
    1.4228    0.4228    0.0895    0.0772    0.1772    0.2439    0.2915    0.3272

arr = 1×7
    2.4228    0.9228    0.4228    0.1728    0.0228    0.0772    0.1486

arr = 1×6
    3.4228    1.4228    0.7561    0.4228    0.2228    0.0895

arr = 1×5
    4.4228    1.9228    1.0895    0.6728    0.4228

arr = 1×4
    5.4228    2.4228    1.4228    0.9228

arr = 1×3
    6.4228    2.9228    1.7561

arr = 1×2
    7.4228    3.4228

arr = 8.4228
```

Notice the highlighted values. When P > Q the algorithm is going to compute values that are always going to be larger than our current bestapprox. To save unnecessary computations and improve efficiency, we need to skip all the computations of $\left|\frac{p}{q} - \gamma\right|$ where $\frac{p}{q} \geq 1$. To do this we need to rethink our strategy.

First, we will calculate as a base case, before entering the while loop. In the base case, p = 1 and q = 2. Notice that the entries in yellow where $\frac{p}{q} \geq 1$, therefore P+1. Changing the boundaries to compute from P+1 to N-P, takes out all the unnecessary calculations. Note that it is P+1 and not P, as we want to eliminate the entries where p=q, which the base case covers.

```
% Base  case
arr = abs(P./(1:N-P) - emconst)
p = 1; q = 2;
```

```
 while P <= N-1 && length(arr) > 2 % length(arr) > 2 stops iterating over empty
arrays

     arr = abs(P./(P+1:N-P) - emconst) % from P+1, as already checked the case
when p = q in base case, to N-P.
     smallest = min(arr);

     if smallest < bestapprox % Update bestapprox, p and q. If better estimate.
         bestapprox = smallest;
         p = P;
         q = find(arr == min(arr)) + P; % find(arr == min(arr)) finds the
position in array with the smallest value, + P to account for the values we took
away from the beginning of the array.
     end
     P = P+1;
 end
```

```
arr = 1×9
    0.4228    0.0772    0.2439    0.3272    0.3772    0.4105    0.4344    0.4522    0.4661

arr = 1×8
    0.0772    0.2439    0.3272    0.3772    0.4105    0.4344    0.4522    0.4661

arr = 1×6
    0.0895    0.0772    0.1772    0.2439    0.2915    0.3272

arr = 1×4
    0.1728    0.0228    0.0772    0.1486

arr = 1×2
    0.2228    0.0895
```

To stop iterating over empty arrays, check if the length of the array is greater than 2. This takes less computational overhead than ~isempty. This strategy has significantly reduced the number of computations and iterations.

In this next section[3], I measure the speed of this function by using MATLAB inbuilt functions tic toc. I run this 100 times and then take the average by using the mean() function.

```
 elapsed_time = zeros(100,1) ;  % initiliaze the elapsed times

 for i = 1:100
     t = tic;
     [p,q] = AppEm(2021);
     t = toc(t);
     elapsed_time(i) = t;
 end
```

[3]The MathWorks, Inc., get average elapsed time, {Retrieved March 09, 2021},, <https://uk.mathworks.com/matlabcentral/answers/317907-get-average-elapsed-time>

```
time_avg = mean(elapsed_time)
```

```
time_avg = 0.0039
```

Using the same code but substituting the other two function yields time_avg = 0.01243 for MyLuckynum(2021) and time_avg = 0.37604 for Beautisqnum(360322021).

2 MyLuckynum.

This question asks to find the first occurrence of a MyLuckynum which must be greater than or equal to the input, N.

A MyLucky number, n, is the following:

- A MyLucky number cannot be prime.
- It has only odd prime factors. This means n has no prime factors equal to 2.
- All the prime factors are distinct. This means there are no duplicate primes factors.
- For every prime factor, p. There must be no remainder after the operation $\frac{n-1}{p-1}$. i.e. every $p-1$ divides $n-1$.

Example: 10585 is a MyLucky number. Its prime factors $5,\ 29,\ 73$ are all odd, distinct, and

$$\frac{10585}{5-1} = 2646 \quad ; \quad \frac{10585}{29-1} = 378 \quad ; \quad \frac{10585-1}{73-1} = 147.$$

Approach

1. Use a while loop to iterate from N until the first My Lucky number.
2. When the first MyLucky number, n, is found break the loop and return n.

During the iterative process, if a condition is not met then the algorithm moves on to the next iteration. This is achieved with by incrementing N and using the keyword continue.

Uses a while loop, which breaks at the first occurrence of a MyLucky number.

1. Check if N is prime. If this is the case, increment N and move on to next iteration using continue. If N is prime, the only prime factor is 1 and itself. (There is no use in checking any other condition beforehand, this saves computational overhead).
2. Find the factors of N using the factor() function. Store them in a variable named factors.
3. Check to see that the prime factors are odd. The only even prime factor is 2, so check if factors == 2 using the find() function. If 2 is a prime factor, the algorithm increments N and continues.
4. Check that the factors are distinct. If they are not, increment N and continue. This is achieved[4] by checking the length of the factors vector against the length of the factors

[4] The MathWorks, Inc., Check if there are repeated elements in a vector, {Retrieved March 01, 2021}, <https://uk.mathworks.com/matlabcentral/answers/55712-check-if-there-are-repeated-elements-in-a-vector>

vector after passing it through the function unique(). The unique() function returns the input array but with no repetitions.

5. Finally, check that there is no remainder from N-1 divided by the factors-1. If not, increment N.

```
 while N>0

         if isprime(N) % we do not want to check the prime numbers as they have
no prime factors beside themselves.
             N = N +1;
             continue
         end

         factors = factor(N);     % factors of N.

         if find(factors == 2)    % check the prime factors are odd. The only
even prime factor is 2.
             N = N +1;continue
         elseif length(factors) ~= length(unique(factors)) % check that there
are no duplicate prime factors.
             N = N +1;
             continue
         end

         a = N-1;
         b = factors - 1;

         comp = rem(a,b);  % remainder of N-1 by factors-1.

         if comp == 0      % check that there is no remainder from N-1 divided
by the factors-1.
             n = N;
             break
         else
             N = N +1;   % If there is, then increment N.
         end
     end
```

If all the cases above are met, the algorithm has found a Lucky number. Return n = N.

3 Beautisqnum

This function takes an input N and finds the closest n^2 to N such that:

- $n^2$is a perfect square number.
- All the digits of n^2 are distinct, from 1 to 9 (note that zero is not included).

Beautisqnum then returns n, the square root of n^2. The input N is assumed to be between 10^8 and $10^9 - 1$.

Approach

This algorithm uses a for loop which iterates between $\sqrt{10^8}$ and $\sqrt{10^9}$. This insures one checks the numbers that, when squared, are in the range 10^8 and $10^9 - 1$. The adoption of the ceil() and floor() function could be used here, but as it yields the same result without them, no need to complicate things.

Calculate the square of x in a variable named *squared*, then use the numtodig() function[5] to store the digits of that x. The loop goes through this process on each iteration. numtodig() is a function stored in the same directory. It takes an input n and returns an array of the constituent digits of that number.

```
for x = sqrt(10^8):sqrt(10^9)
    squared = x^2;                  % the square
    digits = numtodig(squared);     % the digits of said square
end
```

Next check if the digits are unique i.e there are no repeating digits. Use the function unique() as we did before, but comparing the length of the number of digits. Check if there is a zero in the list of digits. If so, *continue* the loop. If these conditions are met, add these numbers to an initialized variable named beaut_nums to store the beautiful numbers.

```
beaut_nums = [];            % initialize to an array.

for x = sqrt(10^8):sqrt(10^9)
    squared = x^2;
    digits = numtodig(squared);

    if length(unique(digits)) == length(numtodig(digits))   % checks if the
digits do not repeat.

        if find(digits == 0)   % skip if there is a digit equal to 0.
            continue
        end
        beaut_nums = [squared beaut_nums];
    end
end
beaut_nums
```

```
beaut_nums = 1×30
```

[5] Prof Jitesh Gajjar, labdemo5 – number to vector of digits, {2020-2021}, <https://online.manchester.ac.uk/bbcswebdav/pid-12215832-dt-content-rid-66441489_1/courses/I3133-MATH-36032-1201-2SE-009260/labdemo5.html>

```
923187456   847159236   842973156   743816529   735982641   714653289 ⋯
```

The beaut_nums shows all the beautiful numbers in the given range. The problem with the following code is that beaut_nums changes size on every iteration, this is bad practice, see reference[6]. Instead, allocate the variable beforehand. This improves performance and speed.

For this reason, it is worth noting that the number of beautiful numbers that there are in this range is 30. Therefore, if beaut_nums is initialized to a row vector of ones from 1 to 30, all that is required is to mutate this array, using a counter.

```
beaut_nums = ones(1, 30);       % preallocate beaut_nums.
i = 0;                          % counter to track number of beautiful numbers.

for x = sqrt(10^8):sqrt(10^9)
    squared = x^2;
    digits = numtodig(squared);

    if length(unique(digits)) == length(numtodig(digits))

        if find(digits == 0) % skip if there is a digit equal to 0.
            continue
        end

        i = i + 1;                  % increase counter when a beautiful number
is found.
        beaut_nums(i) = squared;    % mutate the initialized variable
beaut_nums.
    end
end
```

The final few steps, outside the for loop, involve the following:

```
diff = abs(beaut_nums-N); % take the absolute value of the difference between
% the beautiful numbers and N.

pos = diff == min(diff);            % find the position of the smallest
difference, to get the closest value to N.

n_squared = beaut_nums(pos);        % pos variable is the position of the
closest n^2 to N.
n = sqrt(n_squared);                % return n, the beautiful square number.
```

[6] John D'Errico (2021). Incremental growth of an array, revisited, MATLAB Central File Exchange. {Retrieved March 09, 2021}, <https://www.mathworks.com/matlabcentral/fileexchange/8334-incremental-growth-of-an-array-revisited>,