Question 1

Cubic Taxicab number

Problem

Cubic Taxicab number: Is a positive integer which can be written in two or more distinct ways of the form:

$$t = a^3 + b^3$$

where a, $b \in \mathbb{Z}^+$

Write a function `CubicTaxicabNum(N)` that takes an input N and returns the smallest cubic taxicab number which is greater than or equal to N.

1.1 Approach

Suppose we have a function `isCubicTaxiCab(X)` which returns a `boolean` value determining whether $X \in \mathbb{Z}^+$ is a taxicab number. We will assume the input N is always a positive integer. Using this function we can find the smallest cubic taxicab number greater than or equal to N by checking each integer greater than or equal to N, until we find a cubic taxicab number.

```
function ctn = CubicTaxicabNum(N)
% CUBICTAXICABNUM returns the smallest cubic taxicab number greater than
% or equal to N

ctn = N;
while (~isCubicTaxiCab(ctn)) % Until we find a cubic taxicab number.
    ctn = ctn + 1;
end
end
```

Now we can implement the function `isCubicTaxiCab(X)`. First, we make an observation about the solution. Assume $t \in \mathbb{Z}^+$ is a cubic taxicab number. Then we have the following observation:

Observation: If $t = a^3 + b^3$ for $a, b \in \mathbb{Z}^+$ then assuming without loss of generality that $a \leqslant b$ we have:

$$a \leqslant b \leqslant \texttt{floor(}\sqrt[3]{t}\texttt{)}$$

where $\texttt{floor(}\sqrt[3]{t}\texttt{)}$ is the truncated value of the cube root of t.
e.g. $t = 20 \implies \texttt{floor(}\sqrt[3]{20}\texttt{)} = \texttt{floor(2.714...))} = 2$.

It follows from the observation that it is sufficient to check numbers in the range $[1,$ `floor(`$\sqrt[3]{t}$`)`$]$ as possible candidates for a and b such that $t = a^3 + b^3$. We claim the following is a solution for the function `isCubicTaxiCab(X)`:

```
1 function x = isCubicTaxiCab(X)
2 % ISCUBICTAXICAB returns a boolean value determining if X is a cubic
3 % taxicab number or not.
4
5 i = 1; j = floor(nthroot(X, 3));
6 comb_count = 0;
7 A = 1:j;
8 x = false;
9 combo = zeros(2); % Tracks the first two combinations if x is ctn
10 while (i < j && comb_count < 2)
11     cube_sum = A(i)^3 + A(j)^3;
12     if (cube_sum > X)
13         j = j - 1;
14     elseif (cube_sum < X)
15         i = i + 1;
16     else
17         comb_count = comb_count + 1;
18         combo(comb_count,:) = [i j];
19         i = i + 1; j = j - 1;
20     end
21 end
22 if (comb_count == 2)
23     x = true;
24 %     disp(combo); % uncomment to see the first 2 sum of cubes.
25 end
26 end
```

1.2 Analysis

1.2.1 Correctedness:

Let $Y =$ `floor(`$\sqrt[3]{X}$`)`. The above function attempts to find two distinct combinations from the range $A = [1, Y] \subset \mathbb{Z}^+$ for which the sum of cubes is equal to X.

The algorithm first checks if $1^3 + Y^3 = X$ i.e $A(1)^3 + A(Y)^3 = X$. If this is the case, then we have found one combination whose sum of cubes is equal to X (line 16-19). However, if the sum of cubes is greater than X then we must add a smaller value to 1^3 to get closer to X. Hence, we then decrement j by 1 (line 13). With a similar argument, if the sum of cubes is less than X then we must increment i by one (line 15).

This is repeated until either $i = j$ or we have found 2 combinations (condition in the `while` loop). The latter implies that X is a cubic taxicab number, whereas if $i = j$ before `comb_count` $= 2$ then we can conclude that X is not a cubic taxicab number (line 22-24) because there are no other valid possible combinations to check in A

The algorithm checks all possible combinations in the list $A = [1, Y] \subset \mathbb{Z}^+$. Therefore, we will always find two combinations if X is a cubic taxicab number and will only return `false` when it isn't. Hence, the algorithm is correct. $\square$

1.2.2 Efficiency:

`CubicTaxicabNum(N)`:

This function clearly iterates n times where n is the difference between N and the smallest cubic taxicab number greater than or equal to N.

`isCubicTaxiCab(X)`:

This function loops at most $Y =$ `floor(`$\sqrt[3]{X}$`)` times, in the case when X is **not** a cubic taxicab number. The algorithm checks all possible combinations in the list $A = [1, Y] \subset \mathbb{Z}^+$ and terminates without finding at least two correct combinations.

This implementation is much more efficient than a "brute force" approach where we would check every possible combination in A which can happen at most Y^2 times.

We can say this solution runs in *"linear time complexity in Y"* which is significantly faster than the "brute force" approach which is *"polynomial time complexity in Y"*, especially for very large input X.

1.3 Results

Below are the following results for two inputs $N = 1$ and $N = 36032$.

```
>> CubicTaxicabNum(1)
     1    12
     9    10
ans = 1729
```

N=1

This is a correct result because 1729 is the first cubic taxicab number, associated with an anecdote about Ramanujan by G. H. Hardy.

$$1729 = 1^3 + 12^3 = 9^3 + 10^3$$

NB: The above result is run by uncommenting line 24 in the `isCubicTaxiCab(X)` function above. This also outputs the first two combinations whose cube sum is equal to the cubic taxicab number found.

```
>> CubicTaxicabNum(36032)
     2    34
    15    33
ans = 39312
```

N=36032

Therefore, $\mathbf{39312} = 2^3 + 34^3 = 15^3 + 33^3 = ...$ (possibly more) is the smallest Cubic Taxicab number greater than or equal to 36032

Question 2

Catalan's Constant

Problem

Here we have the Catalan's constant:

$$G = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)^2} = \frac{1}{1^2} - \frac{1}{3^2} + \frac{1}{5^2} + ... + \approx 0.915965594177219$$

G can be expressed in terms of various sums and series or special integrals, however it hasn't yet been proven to be irrational or not.

Write a function `RatAppCat(N)` which takes an input $N \in \mathbb{Z}^+$ and returns a pair (p, q) such that $p, q \in \mathbb{Z}^+$ and p/q is the best rational approximation of G. i.e. $|p/q - G|$ is the smallest. We add a constraint that $p + q \leqslant N$.

2.1 Approach

Observation: For a given $q \in \mathbb{Z}^+$, we can find the *accurate* $p^* \in \mathbb{R}^+$ such that $p^* = qG$. Then for any such p^*, $p = \texttt{round}(p^*) \in \mathbb{Z}^+$ will give the best approximation of $p/q \approx G$ for the given q.

Moreover, it is trivial to see that if $p_0 + q_0 > N$ then, $p_1 + q_1 > N$, where $p_0, p_1, q_0, q_1 \in \mathbb{Z}^+$ such that p_0/q_0 and p_1/q_1 are approximations of G with $p_0 \leq p_1$ and $q_0 \leq q_1$.

Using the first observation it is sufficient to iterate over all the values of $q \in [1, N]$ such that $p + q \leq N$ where $p = \texttt{round}(qG) \in \mathbb{Z}^+$. On each iteration, we can calculate $d = |p/q - G|$, whilst keeping track of the smallest d. Then using the second observation, we can stop iterating over q any further when we get the first $p + q > N$.

The following function `RatAppCat(N)` implements the above approach:

```
function [p, q] = RatAppCat(N)
% RATAPPCAT The best rational approximation p/q of the Catalan 's constant,
% among all pairs of (p,q) such that p+q<=N

G = double(catalan);
min_dif = 1;

for qi = 1:N
    rvp = round(G*qi); % rounded value of 'accurate' decimal p*
    if (rvp + qi > N)
```

```
        return; % if we find a p+q > N then we are done
    end
    dif = abs(rvp/qi - G);
    if (dif < min_dif)
        min_dif = dif;
        p = rvp; q = qi;
    end
end
end
```

2.2 Analysis

2.2.1 Correctedness:

The algorithm loops from `qi = 1 to N`. For each `qi` the *rounded perfect value* `rvp = round(G*qi)` is calculated, which we know will be the best value of p for the given `qi` from our observation. The algorithm then checks if the current `rvp + q > N`. In the case where this is true, the algorithm halts because we can no longer find a better solution whilst obeying our restriction that $p + q \leq N$.

However, if this condition is false, then $d = |p/q - G|$ is calculated (for `qi`) and if this is smaller than any previous d, then we pick this combination to be the best p and q.

□

2.2.2 Efficiency:

The algorithm approximately loops at most $N/2$ times because for a given q; $p = \texttt{round}(qG)$ will be "very close" to q as $G \approx 0.91596559417721$. Therefore, if `qi` $\approx N/2$ then `qi + rvp` $\approx$ `qi + qi` $\approx 2 \times N/2 = N$ then `qi + 1 + rvp` $\gtrapprox N$.

Therefore, the algorithm runs in *"linear time complexity in N"*, however, it does approximately $N/2$ operations rather than N.

2.3 Results

Below is the result for $N = 2018$.

```
>> [p,q] = RatAppCat(2018)
p = 109
q = 119

>> p/q
ans = 0.915966386554622
```

N=2018

Question 3

Sum of reciprocal squares with prime factors

Problem

Consider the sum of reciprocal squares:

$$\sum_{k=1}^{\infty} \frac{(-1)^{\Omega(k)}}{k^2} = \frac{1}{1^2} - \frac{1}{2^2} - \frac{1}{3^2} + \frac{1}{4^2} + \dots \tag{3.1}$$

where $\Omega(k)$ is the number of prime factors (with multiplicity) of k and $\Omega(1) = 0$. e.g. $\Omega(p) = 1$ for any prime p; $\Omega(4) = 2$ because $4 = 2 \times 2$.

Find a reasonable approximation for the value of the above series by truncating a finite number of terms. Analyse the accuracy of the answer (i.e the number of correct decimal digits) based on computation with no more than 1,000,000 truncated terms.

3.1 Approach

Observation: First we observe the *Basel Problem* which was solved by Leonhard Euler. The Basel Problem is the infinite sum of reciprocals, which has the exact value $\pi^2/6$.

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6} \tag{3.2}$$

Now (3.1) can be written as:

$$\sum_{k=1}^{\infty} \frac{(-1)^{\Omega(k)}}{k^2} = \sum_{k=1}^{n} \frac{(-1)^{\Omega(k)}}{k^2} + \sum_{k=n+1}^{\infty} \frac{(-1)^{\Omega(k)}}{k^2}$$

Similarly (3.2) can written as:

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = \sum_{k=1}^{n} \frac{1}{k^2} + \sum_{k=n+1}^{\infty} \frac{1}{k^2}$$

Now it is very easy to see that:

$$\sum_{k=1}^{\infty} \frac{(-1)^{\Omega(k)}}{k^2} \leq \sum_{k=1}^{\infty} \frac{1}{k^2}$$
$$\implies \sum_{k=1}^{n} \frac{(-1)^{\Omega(k)}}{k^2} + \sum_{k=n+1}^{\infty} \frac{(-1)^{\Omega(k)}}{k^2} \leq \sum_{k=1}^{n} \frac{1}{k^2} + \sum_{k=n+1}^{\infty} \frac{1}{k^2}$$

Therefore, we have that the error of approximating n terms for (3.1) is at most the error of approximating n terms for (3.2). Since, we know the exact value for (3.2), we can find the exact value of the error of approximating n terms for (3.2) i.e:

$$\sum_{k=n+1}^{\infty} \frac{(-1)^{\Omega(k)}}{k^2} \leq \sum_{k=n+1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6} - \sum_{k=1}^{n} \frac{1}{k^2} \tag{3.3}$$

For a significantly large n, the error will be significantly small and we can use this to bound $\sum_{k=1}^{n} \frac{(-1)^{\Omega(k)}}{k^2}$ from above and below as follows:

$$\sum_{k=1}^{\infty} \frac{(-1)^{\Omega(k)}}{k^2} - \sum_{k=n+1}^{\infty} \frac{1}{k^2} \leq \sum_{k=1}^{\infty} \frac{(-1)^{\Omega(k)}}{k^2} \leq \sum_{k=1}^{\infty} \frac{(-1)^{\Omega(k)}}{k^2} + \sum_{k=n+1}^{\infty} \frac{1}{k^2} \tag{3.4}$$

If the value of the lower bound is equal to the upper bound up to some tolerance (i.e. some number of decimal digits), then we can be sure that (3.1) is accurate up to the same tolerance.

The function below calculates exactly the value of (3.1) up to some tolerance by truncating no more than 1,000,000 terms.

```
function SumPF
% SUMPF find an approximation of the sum of reciprocal squares with prime factors

N = 1000000;
sum = 1; % Value of the series upto N terms
basel_sum = 1; % Value of basel problem upto N terms
upper_b = 0; % Upper bound of the value of the series for k terms
lower_b = 0; % Lower bound of the value of the series for k terms
tolerance = 0;
result = [];

for k = 2:N
    sum = sum + (((-1)^length(factor(k)))/k^2); % add terms for our problem (3.1)
    basel_sum = basel_sum + 1/k^2; % add terms for the Basel problem (3.2)
    basel_err = ((pi^2)/6 - basel_sum); % error of the basel problem (3.3)
    upper_b = sum + basel_err;
    lower_b = sum - basel_err;
    % Increase the number of decimal places till the lower and upper bound are equal.
    while (round(lower_b, tolerance) == round(upper_b, tolerance))
        result = [sum; round(sum, tolerance); tolerance; k;];
        tolerance = tolerance + 1;
    end
end
disp(result(1));
disp("Value = " + num2str(result(2)));
disp("Accuracy = " + num2str(result(3)) + " decimal digits");
disp("Number of truncated terms = " + num2str(result(4)));
end
```

3.2 Analysis

3.2.1 Correctedness:

`sumPF = 1` and `basel_sum = 1` at the start which is the first term in both series. The algorithm then calculates the value of both (3.1) and (3.2) up to $N = 1000000$ terms. For each term that is added, the upper and lower bounds are calculated as defined in (3.4). `tolerance` keeps a track of the number of decimal digits that are accurate for `sumPF`. If the lower bound and upper bound are "close enough" (i.e. equal up to the tolerance), we can conclude that the approximated value of (3.1) is accurate up to that many decimal digits.

Once the accurate value of $\texttt{tolerance}_i$ is found, the algorithm increments the value of `tolerance` to $\texttt{tolerance}_{i+1}$ (i.e. increase the decimal digits). By the end of the for loop, `result` will store the value of (3.1) for the first $X(\leq N)$ terms which is accurate to the `tolerance` value.

□

3.2.2 Efficiency:

The for loop is executed $N - 1$ times. On each iteration of the for loop, the while loop can be executed a finitely arbitrary amount of times depending on "how accurate the value for the first k terms is". As N increases the algorithm has to perform more operations. Therefore, the algorithm runs in "linear time complexity in N" at best.

3.3 Results

The function prints the value, accuracy and number of truncated terms. Below is the output for running the program with $N = 1000000$.

```
>> SumPF()
   0.657973627437163

Value = 0.65797
Accuracy = 5 decimal digits
Number of truncated terms = 728565
```

N=1000000

The result confirms that

$$\sum_{k=1}^{\infty} \frac{(-1)^{\Omega(k)}}{k^2} \approx \mathbf{0.65797}3627437163$$

is **accurate to 5 decimal digits**. We can also see that the number of terms that provide a value correct to 5 decimal digits is 728565. Therefore, running the algorithm with $728565 \leq N \leq 1000000$ will return the same result. To improve the accuracy of the result, we can increase N. However, this will increase computation.