1 Temperature Converter

```matlab
function C = tocelcius (F)
% Converts temperature values from Fahrenheit scale to Celcius scale
C = 5/9*(F - 32);
end
```

```matlab
function F = tofahrenheit (C)
% Converts temperature values from Celcius scale to Fahrenheit scale
F = 9/5*C + 32;
end
```

Many MATLAB functions work on array input variables. It is a good practice if your functions follow a similar approach. Since our conversion formula is simple the operators work on both single values and arrays. If you run the following lines in command window, they should work properly.

```matlab
icemelts = tofahrenheit(0) % should echo 32
ifreeze = tocelcius(0) % should echo -17.778

ithaca_monthly = tocelcius([20.2,22.2,31.1,45.8,58.5,66.1,...
71.9,67.5,63.0,50.2,44.9,33.2])

london_celcius = [7.9,8.2,10.9,13.3,17.2,20.2,22.8,22.6,19.3,15.2,10.9,8.8];
london_monthly = tofahrenheit(london_celcius)
```

2 Sum of Primes

You can use the function `isprime` to check whether a number is a prime or not, and the function `primes` to get an array of all primes smaller than or equal to a number. You can use a loop to sum only the numbers which are prime. Or you can use the array returned by `primes` as a range. You can also use the built-in `sum` function. There are various ways this task can be accomplished. Some of them might take a longer time. Why? You can use functions `tic` and `toc` to measure the time spent on a code fragment. See the appendix in lecture slides for July 9.

```matlab
N = 123456; s = 0;
for j=1:N
    if isprime(j),
        s = s + j;
    end
end
fprintf('Sum of primes smaller than %d is %d
',N,s);
```
N = 123456;
s = sum(primes(N));
fprintf('Sum of primes smaller than %d is %d
',N,s);

s = 0;
for j = primes(N)
    s = s + j;
end
fprintf('Sum of primes smaller than %d is %d
',N,s);

3 Capitalize

Lowercase and uppercase letters are separated by a fixed value, 32. You can take a look at the ASCII table to see the range of values for lowercase letters. Functions double and char can be used to convert between characters and numbers. When you capitalize the string, you have to be careful to limit your changes only to the lowercase letters; punctuation, digits, uppercase letters etc. should remain untouched.

function capstr = capitalize(str)
% Returns a copy of the original string, after converting lowercase % letters to uppercase.

capstr = str;
shift = 'A' - 'a';
for j = 1:length(str)
    if 'a' <= str(j) && str(j) <= 'z'
        capstr(j) = char(str(j) + shift);
    end
end

A good strategy is to write a script file to test the functions you are writing. In this script you can write tests based on how a correct function should behave. For example we expect our function to perform equivalent to MATLAB function upper. We can compare our results with those returned by upper. String comparison can be accomplished by the MATLAB function strcmp. An example testing script could be as follows:

% it should convert a lowercase word into uppercase
test1 = 'hello';
expect1 = upper(test1);
result1 = capitalize(test1);
assert(strcmp(result1,expect1));

% it should leave capital letters as they are
test2 = 'WORLD';
expect2 = upper(test2);
result2 = capitalize(test2);
assert(strcmp(result2,expect2));

% it should not change digits
test3 = '0123456789';
expect3 = upper(test3);
result3 = capitalize(test3);
assert(strcmp(result3,expect3));

% it should not change punctuation characters etc.
test4 = '![]}{#%ˆ&*()−+=˜'';:"@/\';
expect4 = upper(test4);
result4 = capitalize(test4);
assert(strcmp(result4,expect4));

% you can add as many tests as you like. you can also create
% summary statistics using the number of tests passed and failed.

4 Estimate Pi

You have to use larger numbers for \(N\) to get better approximations. Notice that we didn’t have to store the values in arrays. We could have generated a single point during each iteration of the loop. In this problem having the coordinates stored in the arrays helps us to plot them in one shot (later in the course).

% In this script we will estimate value of pi
% using a numerical experiment. After generating
% uniformly distributed random points inside a square
% region, we count the number of points falling
% inside the inner tangent circle. The ratio
% of number of points inside the circle to the total
% number can be used to find an estimate for pi.

N = input('Enter the number of points: '); % Take user input
x = 2*rand(N,1)−1; % Generate random x coordinates
y = 2*rand(N,1)−1; % Generate random y coordinates
n = 0; % Count the number of points
for j=1:N
    if x(j)^2 + y(j)^2 < 1 % inside the circle?
        n = n + 1;
    end
end

est_pi = 4*n/N; % Compute the estimate
rel_err = abs(est_pi−pi)/pi; % and the relative error
fprintf('Estimate for pi is %.15f with relative error %.6f\n', ...
        est_pi, rel_err);
5 Secret Number : 1729

This problem was first asked by Fermat more than three centuries ago. However its fame is due to two other mathematicians, G.H. Hardy and Srinivasa Ramanujan.

“I remember once going to see him when he was lying ill at Putney. I had ridden in taxi cab number 1729 and remarked that the number seemed to me rather a dull one, and that I hoped it was not an unfavorable omen. ‘No,’ he replied, ‘it is a very interesting number; it is the smallest number expressible as the sum of two cubes in two different ways.’ ” (Hardy,1940)

Hence these type of numbers are called \textit{taxicab} numbers.

\[ S = x^3 + y^3 = p^3 + q^3 \]

The most intuitive solution for this problem with your current knowledge is to have \( x, y, p, q \) takes values from a range \( 1:n \) for some number \( n \). Using four nested for-loops, we can check if the equality holds, and declare the minimum \( S \) as our secret number. Note that you can continue to your statements on the next line by using ...

\begin{verbatim}
\begin{verbatim}
\n n = 20;
 minS = 2*n^3; % Anything we can find will be less than 2*n^3

for x = 1:n
  for y = 1:n
    for p = 1:n
      for q = 1:n
        if (x^3 + y^3 == p^3 + q^3) && ...
            (x\neq y) && (x\neq q) && (x\neq p) && (p\neq q)
          if minS > (x^3 + y^3) % repeated computation
            minS = (x^3 + y^3);
          end
        end
      end
    end
  end
end

if minS < 2*n^3
  fprintf('Secret number is: %d
', minS);
end
\end{verbatim}
\end{verbatim}

Let’s try to refine this code step by step. There are repeated computations for \( x^3 + y^3 \). It is better to store it in a variable inside the code segment where it won’t change.
n = 20;
minS = 2*n^3;
for x = 1:n
    for y = 1:n
        S = x^3+y^3; % S will not change within the loops for x and y
        for p = 1:n
            for q = 1:n
                if (S == p^3 + q^3) &&...
                    (x==y) && (x=p) && (x=q) && (p=q)
                    if minS > S, minS = S; end
                end
            end
        end
    end
end
if minS < 2*n^3
    fprintf('Secret number is: %d
', minS);
end

The comparisons to require the condition for distinct x, y, p, q values is cumbersome. If only we could arrange our loop ranges so that we didn’t need those comparisons at all.

One of these four numbers will be the smallest, let it be x. Then y will be the largest. If we also assume an ordering for p and q, say p is the smaller of the two. Then p has to be greater than x, and q has to be smaller than y. We can change the order of loop variables, and always start one step ahead of the smaller one. In this way we can get rid of the not equal comparisons. Now there is no way they can be equal.

n = 20;
minS = 2*n^3;
for x = 1:n % assumed x < p < q < y
    for p = x+1:n % notice the start value for the ranges
        for q = p+1:n
            for y = q+1:n
                S = x^3+y^3;
                if (S == p^3 + q^3)
                    if minS > S, minS = S; end
                end
            end
        end
    end
end
if minS < 2*n^3
    fprintf('Secret number is: %d
', minS);
end
Now the temporary computation for $S$ is wrapped up by more loops. We can gain an incremental speed by changing what we store as temporary.

```matlab
n = 20;
minS = 2*n^3;
for x = 1:n
    for p = x+1:n
        for q = p+1:n
            S = p^3+q^3; % ogres are like onions
            for y = q+1:n
                if (S == x^3 + y^3)
                    if minS > S, minS = S; end
                end
            end
        end
    end
end
if minS < 2*n^3
    fprintf('Secret number is: %d
', minS);
end
```

Do we still need to check whether it is a minimum? Given we can find multiple values for $S$, does our loop ordering guarantee to find the minimum possible $S$ first? Well, it doesn’t guarantee, even if it does find the minimum $S$ first. If you found a candidate $S^\ast$ while searching upto $n^\ast$, you can be sure that it’s indeed the smallest if you check for values of $n$ upto $\sqrt[3]{S^\ast} - 1$, and see nothing smaller than $S^\ast$. (Why?) Here is a script to store and display all $S$ values discovered.

```matlab
n = 50; % let’s use a larger range
results = []; % this is an empty array
for x = 1:n
    for p = x+1:n
        for q = p+1:n
            S = p^3+q^3;
            for y = q+1:n
                if (S == x^3 + y^3)
                    results = [results; S, x, y, p, q];
                end
            end
        end
    end
end
disp('========== S values for range n=1:50 ==========');
disp(results)
sorted = sort(results, 1);
disp('========= Sorted S values =====================');
disp(sorted)
% don’t worry about sorting and 2 dimensional results array
% you will understand them next week.
```
It seems like somebody is curious!

Do we need four loops? Since \( x^3 + y^3 \) and \( p^3 + q^3 \) have similar forms, maybe we can use only two loops and be done?

How could we know which values to start with for \( n \)?

Here is another version to solve this problem which automatically searches the range for \( n \). It uses two dimensional arrays and find function. We will learn find function when we are coding the Prime Sieve problem on Wednesday.

```matlab
increment_step = 50;
start = 20;
is_found = 0;
min_sum = Inf;
while ~is_found
    cur_sum = zeros(start,start);
    for x = 1: start
        for y = x : start
            sum_cub = x^3 + y^3;
            [r,c] = find(cur_sum == sum_cub);
            if length(r) == 1
                fprintf ('(S,x,y,p,q) = (%d,%d,%d,%d,%d)
                [r(1),c(1),sum_cub];
                is_found = 1;
                break;
            else
                cur_sum(x,y) = sum_cub;
            end
        end
        if is_found
            break;
        end
    end
    start = start + increment_step;
end
```

Can you spot a hidden nested loop in the code above?