Computer Assignment 1: Model Solution

1. The Maclaurin series of the function e^x is given by

$$e^x = \sum_{k=0}^{\infty} \frac{1}{k!} x^k.$$

How many terms of the above series expansion you need to compute the number e to 5 digits of accuracy?

Solution:

```
Taylor_exp = 0;
n = 0;
while (exp(1) - Taylor_exp) >= 1e-5
    Taylor_exp = Taylor_exp + 1/factorial(n);
    n = n+1;
end
```

The iteration stops at `n = 8`, so 9 terms of the series are needed to reach 5 digits of accuracy: `exp(1) = 2.718281828459045`,

`Taylor_exp =  2.718278769841270`.

2. The `uint8` class corresponds to integers with values ranging from 0 to 255. They are used for storing and manipulating images in a computer. Using the function `floor`, write a code that transforms `rand(1)` to a proportionally equivalent `uint8` variable.

Solution:

```
u = uint8(floor(rand(1)*256));
```

3. Type `3 < 7 > 5` in the Command Window. Explain in detail why you get this result. Add a logical AND to the above statement so that the output is a logical 1 (true), given that 7 is indeed bigger than both 3 and 5.

Solution: `MATLAB` executes the command `3 < 7 > 5` from left to right. Since,

`3 < 7 = logical 1`

is smaller than `5`, `3 < 7 > 5 = logical 0`.

By adding a logical AND in the middle of the expression fixes the problem:

`3 < 7 && 7 > 5 = logical 1`

4. A logical expression which is always true is called a *tautology*. Using `MATLAB`, show that the logical expression P OR ((NOT P) OR Q) is a tautology.

Solution: We should check the expression for every possible combination of logical variables `p` and `q`:

```
p = true; q = false; p || (~p || q )
p = false; q = true; p || (~p || q )
p = false; q = false; p || (~p || q )
p = true; q = true; p || (~p || q )
```

The output is always `logical 1`.

5. Create, in one line of code (without `for` or `while` loops), a matrix of size 100×100 whose elements are 1 above the diagonal, -1 below the diagonal, and 0 along the diagonal. Consider using `MATLAB`'s built-in function `tril` or `triu`.

   Solution:

```
M = tril(ones(100,100)) - triu(ones(100,100));
```

6. Write a function with header `M = myCheckerBoard(n)`, where `M` is an `4n` $\times$ `4n` matrix with the following form:

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \end{bmatrix}$$

   Note that the upper-left element should always be 1 and the upper-right element should always be 0. You can visualize your result by running the command `imshow(M)`. *Hint*: you might find the function `repmat` useful here.

   Solution:

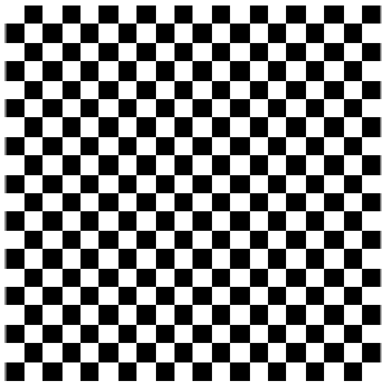
```
function M = MyCheckerBoard(n)
    White = [1 1;1 1]; % the white square
    Black = [0 0; 0 0]; % the black square
    Block = [White Black; Black White]; % the block
       that will be reapeted n times in both directions
    M = repmat(Block , n, n); % create the checker board
end
```

   The function can be tested in a script:

```
n = 10;
M = MyCheckerBoard(n);
imshow(M)
```

7. The equation of an ellipse oriented along the x-y axis with semi-axis of size a and b, respectively, is given by

$$\left(\frac{x}{a}\right)^2 + \left(\frac{y}{b}\right)^2 = 1.$$



In this exercise you will create a function that outputs an `N`-points discretisation of an ellipse of semi-axis lengths a and b, that is rotated at an angle `theta` with respect to the x axis.

Write a function with header `E = ellipse(a, b, theta, N)` following the steps here below.

(i) Using `linspace`, create an array of size 1x`N` consisting of evenly-spaced doubles between 0 and 2π.

(ii) Create an array of size $2\times$`N` where the first/second row is the cosine/sine of the values of the array you created in (i). This will give you an `N`-points discretisation of a circle

(iii) The result of (ii) is an `N`-points discretisation of a circle with radius 1, where the x coordinate is the first row and the y coordinate is the second row. Scale each coordinate by `a` and `b`, respectively, to obtain the discrete version of an ellipse with semi-axis lengths `a` and `b`.

(iv) The matrix

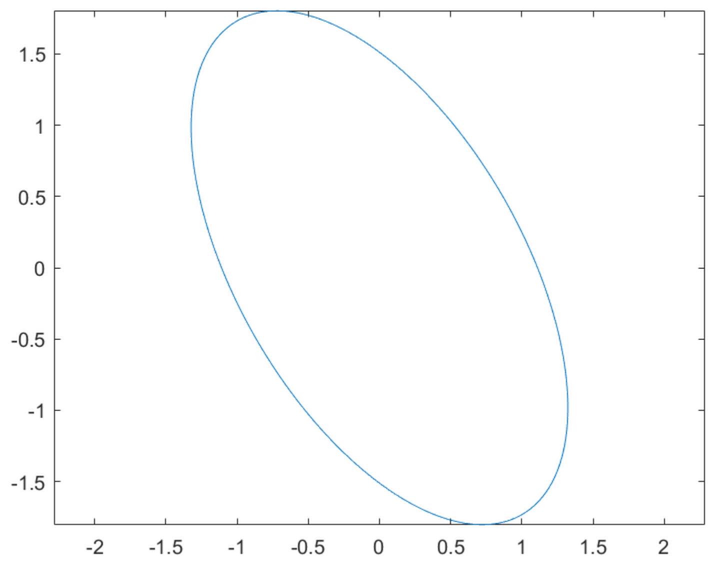
$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

is a rotation matrix – it maps any given vectore $u \in \mathbb{R}^2$ to another vector $v \in \mathbb{R}^2$ such that $\|u\| = \|v\|$ and the angle between the u and v (anticlokwise) is θ. Using this, create an array of size $2\times$ `N` whose columns are θ rotations of the columns of the discretised ellipse of (iii).

You can visualise your results using the command `plot(E(1,:), E(2,:)), axis equal`.

Solution:

```
function E = ellipse(a,b,theta,N)

% create a circle with radius 1
angle = linspace(0,2*pi,N);
circ = [cos(angle); sin(angle)];
```



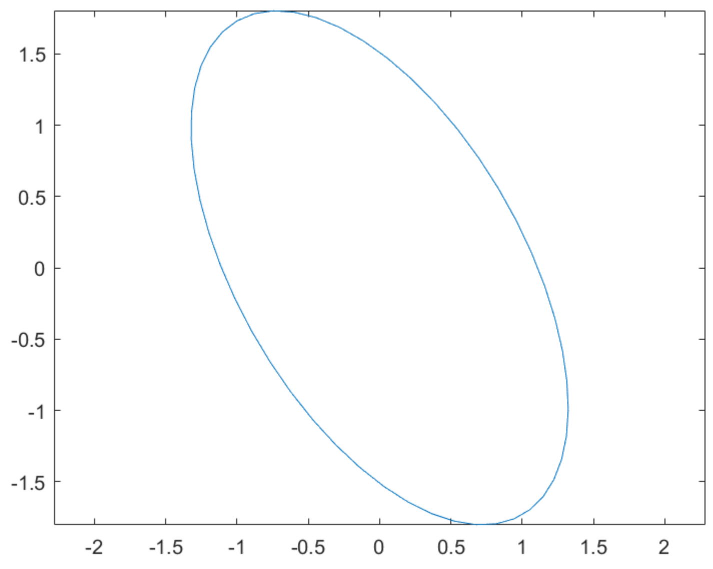
```
% create an ellipse with semi-axis 'a' and 'b'  aligned
    with the x-y axes
E(1,:) = a*circ(1,:);
E(2,:) = b*circ(2,:);

% rotate the ellipse by an angle theta
Rot = [cos(theta) -sin(theta); sin(theta) cos(theta)];
   % build a rotation matrix
E = Rot*E;
end
```

The function can be tested in a script:

```
a = 2;
b = 1;
theta = 2*pi/3;
N = 200;
Ellipse = ellipse(a,b,theta,N);
plot(Ellipse(1,:), Ellipse(2,:)), axis equal;
```

8. Repeat question 7, but now using complex numbers. In particular, instead of using arrays of size 2×`N` to store the discretised circle and ellipses, you will use arrays of size 1×`N`, whose elements are complex numbers to convey the same information. Remember that a complex number multiplied by $e^{i\theta}$ rotates the number an angle θ anti-clockwise. You can visualise your results by running the command `plot(E), axis equal`, where E is the output to your function – your function's output should be an array of complex numbers of size 1×`N`.

   Solution:



```
function E = ellipse_complex(a,b,theta,N)

% create a circle with radius 1
circ = exp(1i*linspace(0,2*pi,N));

% create an ellipse with semi-axis 'a' and 'b'  aligned
    with the x-y axes
E = a*real(circ) + 1i*b*imag(circ);

% rotate the ellipse by an angle theta
E = exp(1i*theta)*E;
end
```

The function can be tested in a script:

```
a = 2;
b = 1;
theta = 2*pi/3;
N = 200;
Ellipse = ellipse_complex(a,b,theta,N);
plot(Ellipse), axis equal;
```

9. Write a program that finds all the prime numbers that are smaller than 100.

   Solution:

```
N = 100;
Primes = [];
for p=1:N
```

```
            isprime = true;
            for k=2:p-1
                if ~mod(p,k)
                    isprime = false;
                    break;
                end
            end
            if isprime
               Primes = [Primes p];
            end
        end
```

10. Download the image `Maxwell_color.png` (see Fig. 1) from Blackboard and save it in your file directory. Run the command `Maxwell = imread(['Maxwell_color.png'])` to read the image in `MATLAB`. The variable `Maxwell` should be a multidimensional array of class `uint8`, of size `n`$\times$`m`$\times$3 (`n` and `m` are the dimensions of the image), where the first two index dimensions indicate the indices of each pixel, and the third index indicate one of each the red, green, and blue (RGB) components of each color pixel.

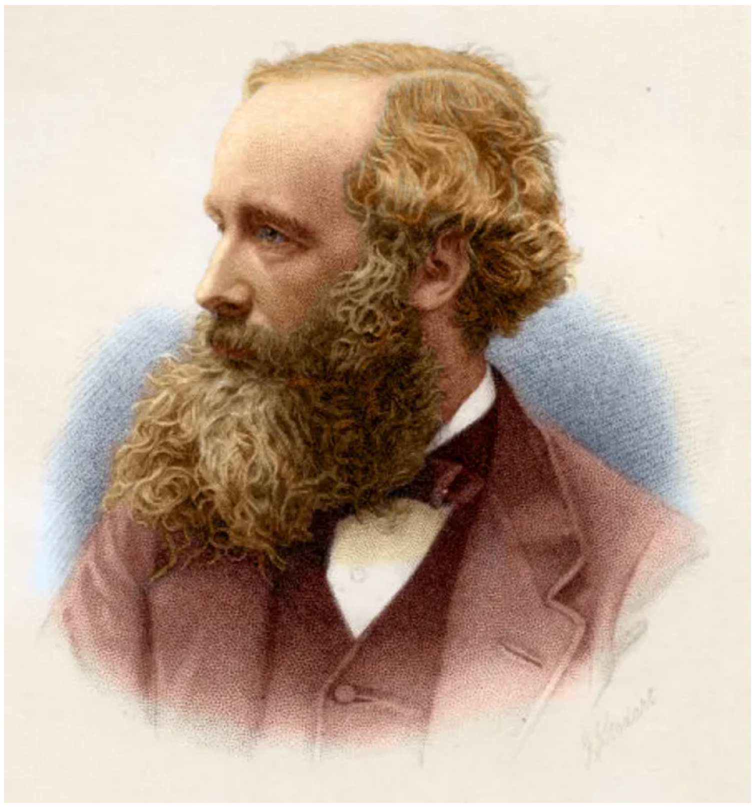


Figure 1: The man on this picture is James Clerk Maxwell, a Scottish Mathematical-Physicist best known for formulating the mathematical theory of electromagnetic waves. James C. Maxwell also did extensive investigations on the nature of color perception, including the RGB description of colors.

(i) From the 3D array `Maxwell`, define three 2D arrays, one for each colour, and transform the types of these arrays from `uint8` to `double`.

(ii) (*scale of greys*) Add the three matrices from (i) in a pondered average with a weight of 0.1 for red, 0.8 for green, and 0.1 for blue. You can visualise your result by running `imshow(uint8(I))`, where `I` is your `double` matrix resulting from the pondered average.

(iii) (*blurring the image*). Given an integer `b` such that 0 <`b`<`m`,`n`, replace every element of `I` by the average its neighbouring pixels, in a neighbourhood of (at most) `b` pixels in both directions. Transform the resulting matrix from `double` to `uint8` and visualise using the function `imshow`.

(iv) (*bonus question*) Write a function that take as an input a colour image and a blurring parameter `b` (the same from (iii)), and outputs a coloured blurred version the image.

Solution:

```
Maxwell = imread(['Maxwell_color.png']);

% Scale of gray immage
Red = double(Maxwell(:,:,1));
Green = double(Maxwell(:,:,2));
Blue = double(Maxwell(:,:,3));
I  =  0.1*Red + 0.8*Green + 0.1*Blue;

% avarage of neighboring pixels
b = 5;
[n,m] = size(I);
I_blur = zeros(n,m); % create a new matrix to store the
    blurred image. This step is necessary.
for i=1:n
    for j=1:m
         B = I(max(1,i-b):min(n,i+b),max(1,j-b):min(m,j
            +b));
         I_blur(i,j) = mean(mean(B));
    end
end
imshow(uint8(I_blur))
```

Bonus question:

```
function I_blur = blurImage(I, b)
[n,m,l] = size(I);
I_blur = zeros(n,m,l); % create a new matrix to store
   the blurred image. This step is necessary.
for i=1:n
```

```
        for j=1:m
            for k=1:l
             B = I(max(1,i-b):min(n,i+b),max(1,j-b):min(m,j
                +b),k);
             I_blur(i,j,k) = mean(mean(B));
            end
        end
    end
    I_blur = uint8(I_blur);
    end
```

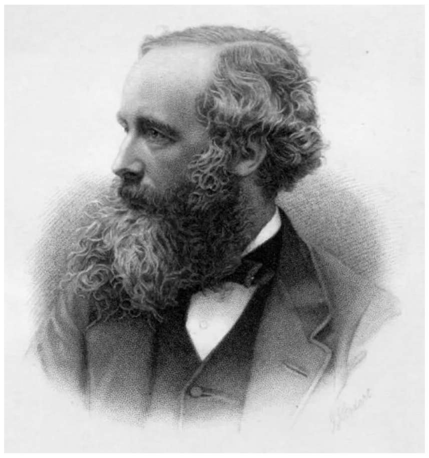
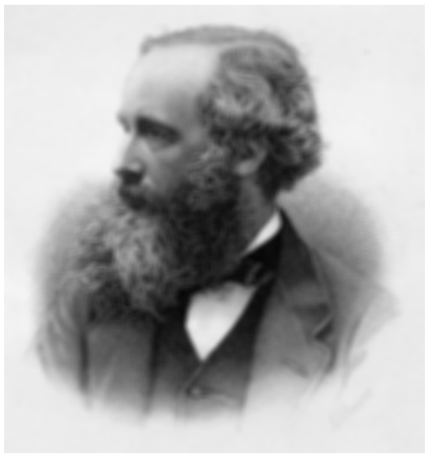
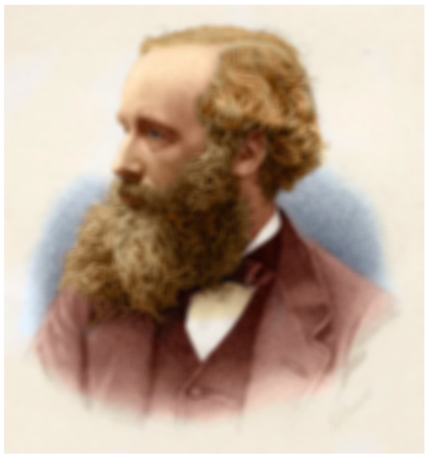




Figure 2: Maxwell's picture in scale of greys with and without blurring, and the original colour picture blurred. $b = 5$.