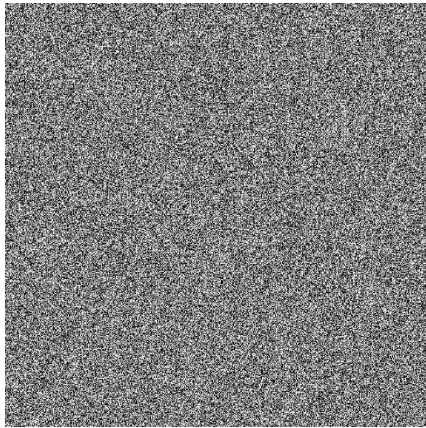


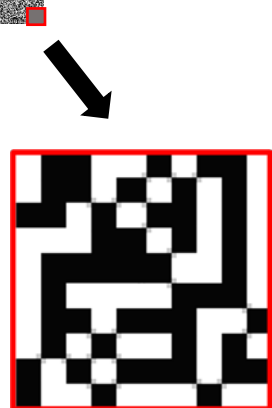
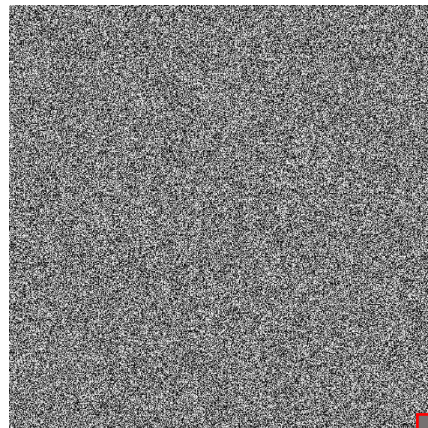
# Visual Cryptography: sending secret messages using images

**Project due date:** You will need to upload your code to MATLAB Grader before  
**11:59 pm on Saturday the 11<sup>th</sup> of September.**

What looks like random noise



A key that also looks like random noise



NOTE: This is version 3 of the project document. If you find a typo please post it to the Piazza project typo thread. If any significant typos (beyond minor grammar and spelling errors) are found, a newer version of the project will be uploaded to Canvas, with those typos corrected.

## Introduction

This project requires you to write code that will encrypt and decrypt messages using a technique known as **visual cryptography**.

Cryptographic algorithms are extremely important and allow messages to be sent securely from one party to another, without having to worry about anyone else being able to read the message if it is intercepted. Many web-based platforms rely on cryptography to send important messages securely. For example, when you want to pay for something online, the online store needs to be sent a message telling them your credit card number, but you don't want anyone else being able to intercept that message (and steal your credit card details) so the message needs to be encrypted.

Cryptographic algorithms work by allowing you to encrypt a message, send it securely and then decrypt it to obtain the original message.

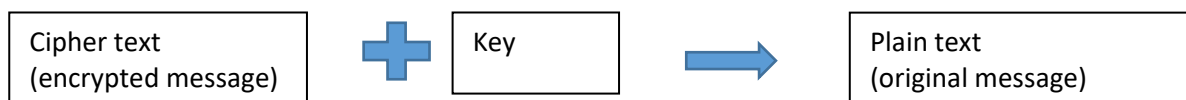
## Encryption

Encryption takes a message (*plain text*) and uses a **key to "lock it up"**, creating an unintelligible message known as *cipher text*. The cipher text can then be sent to someone else without having to worry about other parties reading the message if they intercept it (as they won't be able to understand it).



## Decryption

Decryption is the reverse of encryption, where we take the cipher text and "unlock" the message hidden inside it by using the key. This allows us to read the original plain text message.



Note that decryption requires the use of the key.

## Visual Cryptography

Whereas many cryptographic algorithms are concerned with encrypting and decrypting text, visual cryptography allows us to encrypt and decrypt binary (i.e., black and white) *images*. Here binary refers to having two colours (i.e., black and white), rather than the values 0 and 1.

This means that rather than encrypting *plain text* to produce *cipher text*, we will instead encrypt a **plain image** (with the help of a **key**) to produce a **cipher image**. We can then securely send the image to someone else (who will be able to decrypt the cipher image if they have been previously provided with the key).

Visual cryptography is highly secure, as it is impossible to decrypt a cipher image, unless you have the correct key<sup>1</sup>. A detailed description of exactly how visual cryptography works is provided later in this document. There is also a video explaining the idea too.

### Hidden in plain site

To add an extra layer of security, we can embed a cipher image within another normal image, such as a photo of a cute puppy. This allows us to hide our cipher image unnoticed inside another image. We can then send our secret message to a friend by emailing them our photo of a cute puppy (without anyone suspecting that the image contains a hidden encrypted message).

Similarly, we could give our friend a thumb drive with bunch of key images on it, that are embedded within normal looking images. This is a good idea as if the thumb drive was accidentally left behind in a lab, no-one would know that the images on the drive were key images.

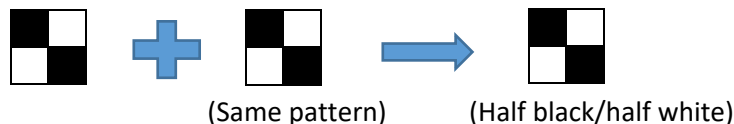
## An overview of visual cryptography

Before we delve into the detail of what functions you will be writing, let's take a quick high-level outline of how messages can be encoded using visual cryptography.

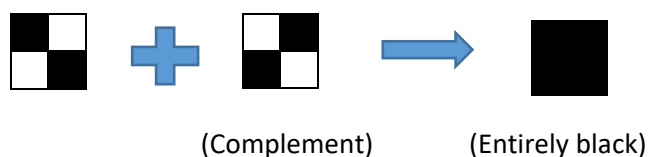
The approach relies on what happens when you place a pattern containing half black and half white pixels on top of the same pattern or the *complement* of that pattern (where black has been swapped for white and white for black).

It might help to think about printing out the patterns using black ink on transparent sheets (where white pixels are left clear) and then lying one sheet on top of another.

e.g.



Verses



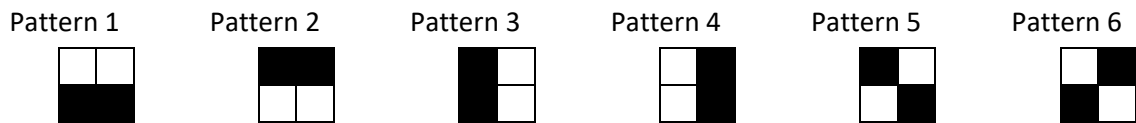
If we place the same pattern on top of the original pattern, the result is a pattern containing half black and half white pixels. From a distance this would look **grey**.

If, however we place the complement of the pattern on top of the original pattern, the result is a pattern containing entirely black pixels. From a distance this would look **black**.

---

<sup>1</sup> Assuming your key is large enough and you use a key only once – if you reuse the same key for sending many images it *might* be possible for someone to reverse engineer the key, if they also know something about the content of the images you are encrypting.

If we limit ourselves to 2x2 patterns, then there are six distinct patterns to choose from that contain half black/half white pixels:

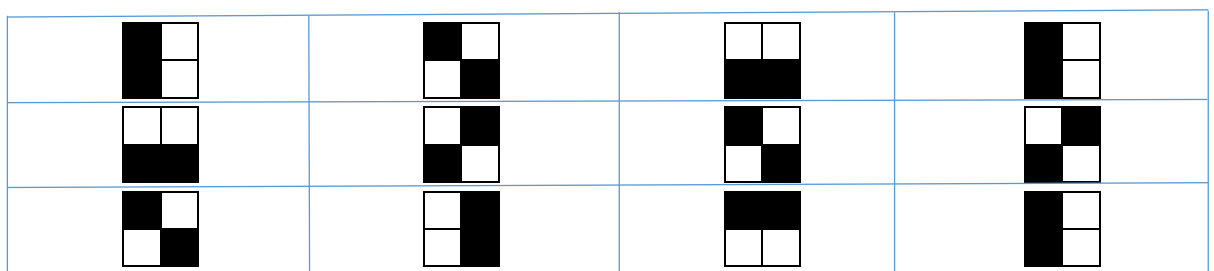


No matter which of the six patterns we choose, placing the pattern on top of itself will yield a pattern of half black/half white pixels (which will look grey from a distance) whereas placing the complement of the pattern on top of itself will yield a pattern containing entirely black pixels (which will look black from a distance).

Armed with this very simple idea, we can now generate a key, encrypt an image, and decrypt it.

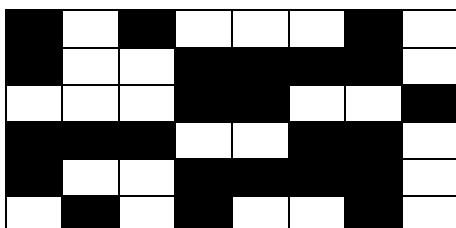
### Step 1: Key creation

First, we create a key array of patterns by assembling a large 2D array of smaller patterns chosen at random (each element of our array will be one of the small 2x2 half black/half white patterns, selected at random from our list of six possibilities). Below is a 2D array of patterns with 3 rows and 4 columns illustrating this idea:



Our key array of patterns will need to have at least as many rows and columns as the image we wish to encrypt. E.g., the above 3 row, 4 column array of patterns could be used to encrypt an image with 3 rows and 4 columns but nothing larger.

When encrypting it is simpler to work with the key array of patterns but if we wish to store the key to use later, we could collapse the above array of patterns into a 6 by 8 array that looks like this:



This could then be written out as an image. Note that whereas our key array of patterns had 3 rows and 4 columns (with each element containing a small 2x2 pattern), our key **image** has 6 rows and 8 columns of pixels (with each element being a single pixel value).

## Step 2: Encryption

Next, we take our plain image (pure black and white images work best) and encrypt it using our key array of patterns and a very simple rule.

If a pixel from our plain image is dark (e.g. black):

Assign to a cipher array of patterns the **complement** of the pattern from the corresponding position in the key array of patterns

Otherwise:

Assign to a cipher array or patterns the **original** pattern from the corresponding position in the key array of patterns

For example, the pixel in row 1 column 3 of the plain image will be encoded using the pattern in row 1 column 3 of the key array of patterns. In our small example above this was the pattern



This pattern (or its complement) will be assigned to row 1 column 3 of the cipher array of patterns.

This means that every pixel in our image is encoded based on whether it is dark or light, using either the complement of the pattern from the corresponding position in our key array (if it is dark) or the original pattern (if it is light).

If we encode every pixel in our image using this approach we will end up with a key array of patterns and a cipher array of patterns, both of which can be eventually be written out as images (with a little bit of work).

## Step 3. Decryption

If the key and cipher images were printed out on transparencies using black ink (leaving white pixels clear) we can decrypt the image simply by placing one image on top of the other. We will see black where the original image had dark pixels and grey where it was light – revealing the original image.

Rather than printing out the images, we can instead write code to simulate placing one image on top of the other. If two pixels that are stacked on top of each other are both white, then we add a white pixel to the corresponding position in an image, otherwise we could add a black pixel (as one or more black pixels are in the stack). This would give us an image that was equivalent to what we would see if we printed the images out and laid one on top of the other (visually some regions appear black and other regions appear a dark grey). With code we can do a little better though and get an image that looks **exactly** like the original image (but twice the size). This requires only a small modification: If two pixels stacked on top of each other include both a black and a white pixel, then we add a black pixel to the decrypted image, otherwise we add a white pixel. With this change we don't get any grey regions.

Without the key image it is practically impossible to reveal the original image. The only way to view the original image would be to find the key by trying every possible combination of keys, which is not feasible for anything but a truly tiny image (even a 12x12 image has over  $1 \times 10^{112}$  possible keys, which is significantly more possibilities than there are atoms in the universe).

## Overview of code to write

You have been provided with three scripts called `CreateKey.m`, `Encryption.m` and `Decryption.m`

Each of these scripts calls functions that you will write. Once you have written all the functions that a script calls, you can then run that script to perform the associated action (i.e. create a key, encrypt a message, decrypt a message).

While each of the supplied scripts will only work if you successfully write **all** the functions that a script calls, each function is submitted and marked separately, via MATLAB Grader. As every function can be marked independently, it is possible to skip over a function if you are finding it challenging and instead have a go at one of the other functions.

Try and submit as many of the **ten** functions as you can. You don't have to tackle them in any particular order, but I recommend starting with the helper functions, as they are two of the easiest to write and are hence a great warmup.

### Part 1: Helper functions

Functions to write

|                              |  |
|------------------------------|--|
| <code>AlterByOne</code>      | Adds 1 to a uint8 value in the range 0 to 255 inclusive (unless the value is 255 in which case it subtracts 1) |
| <code>ImageComplement</code> | Takes an image and swaps black for white and white for black   |

These two simple helper functions are intended to make it easier to write some of the other more complicated functions, as they perform tasks you are likely to want to do multiple times. You will probably find it helpful to call these functions a few times from one or more of the other functions you write. Note that you do not HAVE to call these helper functions from any of the other functions you write, but you should submit them to earn marks, even if you decide not to call them from your other functions.

### Part 2: Key creation

Functions to write for the `CreateKey` script to work

|   |  |
|---|--|
| <code>CreatePatterns</code>   | Creates a 1D cell array containing six special 2x2 patterns of uint8 values, these patterns will be used for key generation  |
| <code>GenerateKey</code>  | Assembles a 2D cell array of 2x2 uint8 patterns to act as a key, with patterns randomly assigned to the cell array based on the values in a 2D array of random numbers between 1 and 6 |
| <code>PatternsToImage</code><br>(also used in <code>Encryption</code> script) | Takes a cell array of 2x2 uint8 patterns (e.g. the patterns for our key image) and converts it to a grayscale image (a 2D uint8 array) so that it can be displayed and saved           |
| <code>EmbedImage</code><br>(also used in <code>Encryption</code> script)      | Used to embed a grayscale image (e.g. the key image) inside another colour image (a 3D uint8 array representing an RGB image)  |

### Part 3: Encryption

Functions to write for the `Encryption` script to work

|  |   |
|--|---|
| <code>ImageToPatterns</code>   | Convert the key image back into a cell array of 2x2 uint8 patterns so that we can more easily work with the key   |
| <code>EncryptImage</code>  | Encrypt the plain image using the key cell array of 2x2 uint8 patterns, to obtain an encrypted cell array of 2x2 uint8 patterns (the patterns for our cipher image)             |
| <code>PatternsToImage</code><br>(also used in <code>CreateKey</code> script) | Takes a cell array of 2x2 uint8 patterns (e.g. the patterns for our cipher image) and converts it to a grayscale image (a 2D uint8 array) so that it can be displayed and saved |
| <code>EmbedImage</code><br>(also used in <code>CreateKey</code> script)      | Used to embed a grayscale image (e.g. the cipher image) inside another colour image (a 3D uint8 array representing an RGB image)  |

### Part 4: Decryption

Functions to write for the `Decryption` script to work

|                           |   |
|---------------------------|---|
| <code>ExtractImage</code> | Called twice, to extract the cipher image and the key image from the images they were hidden in |
| <code>DecryptImage</code> | Will decrypt the cipher image using the key image   |

In the pages that follow you will find detailed explanations of each of the **ten** functions that you need to write to be able to create a key, encrypt an image and decrypt it.

This document shows a few example calls for each function with, but it is expected that you will create your own test values to test your code (in many cases you can work out the answers on some very small sets of test data by hand, so that you know what you should expect as output).

You have a limited number of submission attempts for each function (**six** attempts per function). This means it is important to thoroughly test your function before submission, so that you don't waste attempts.

This project is designed to take the average student around **15 hours** to code, but past experience has shown that some people spend upwards of 60 hours on it (while others will finish it in just a few hours). You will not know ahead of time how long it will take you, so please don't leave it to the last minute to make a start!



## Remember the golden rule

**Write your own code and don't give your code to anyone.** Friends don't let friends share code.

This project needs to be YOUR work, it is an individual project, not a group project. Copying and/or sharing code is academic misconduct. Please note that every project submission is passed through software that detects plagiarism so if you copy **you WILL be caught**. Unfortunately, every year students discover the hard way that I'm not kidding. Let me repeat,

**DO NOT COPY: YOU WILL BE CAUGHT.**

If you give your code to another student and they copy it, you will both be guilty of misconduct. Copying or supplying code typically results in both the person who copied and the person who supplied the code being awarded a mark of zero for the project, as well as your names going on the academic misconduct register. If you have already been found guilty of academic misconduct on another course or assignment the penalty may be even greater (e.g. a fine or fail of the course).

To help you avoid academic misconduct I have put together a very short best practices guide which you should read.

If you have any queries about what is or isn't academic misconduct, please ask so that I can make sure everyone understands what is acceptable behaviour and what isn't.

## How to tackle the project

Do not be daunted by the length of this document. It is long because a lot of explanation is given as to exactly what each function needs to do.

The best way to tackle a big programming task is to break it down into small manageable chunks. A lot of this work has already been done for you in the form of ten functions to write. Each function has a detailed description of what it needs to do and most of the functions can be written independently of each other (the exception being that you might want to use the helper functions to help you write some of the harder ones). Have a read through the entire document and then pick a function to start on.

Remember you don't have to start with `AlterByOne`, although it is one of the simplest functions. You may prefer to start with one of the other fairly straight forward functions such as `ImageComplement` or `CreatePatterns`.

Several functions require careful thought, particularly those that form the heart of encrypting a message (remember those 5 steps of problem solving!). If you are having trouble understanding how a function should work, remember to work through the problem by hand with a small data set.

Note that I don't necessarily expect everyone to write all the functions. Some of them are relatively straight forward (e.g. `GenerateKey` and `DecryptImage`) while others are a bit trickier (e.g. `EncryptImage`). You can still get a pretty good mark even if you don't complete all the functions.

You will receive marks both for functionality (does your code work?) and style (is it well written?) An "A" grade student should be able to nut out almost everything. B and C grade students might not get a fully working solution. Take heart, even if you only get half of the functions working, you can still get over 50% for the project, as long as the code you submit is well written (since you get marks for using good style).



## What Matlab functions do I need to know?

The entire project can be completed using just the functions we have covered in the course manual, lectures and labs plus one additional function (the `mod` function). If you are not familiar with the `mod` function see the video recording that introduces what it is and how to use it to tell if a number is even. You will likely find the `mod` function useful when writing the `EmbedImage` and `ExtractImage` functions.

## What Matlab functions can I use?

When writing your code, while you don't need to use functions we haven't covered in class, you may wish to do so. As well as being able to use any functions we have covered in class, you can also use any other functions that are part of Matlab's **core** distribution (i.e. not functions from any toolboxes you may have installed). Note that for this project ***the use of functions that are only available in toolboxes is not permitted***. Matlab features a large number of optional toolboxes, that contain extra functions. The purpose of this project is for you to get practice at coding, not to simply call some toolbox functions that do all the hard work for you.

**IMPORTANT:** We have configured MATLAB Grader to only allow the use of **core** functions. Your code won't pass MATLAB Grader tests if you have used toolbox functions that aren't in the core distribution.

A list of some of the core MATLAB functions can be found in the MATLAB command reference appendix at the end of the course manual. If you are uncertain whether a particular function is part of the MATLAB's core distribution or not, type the following at the command line

```
which <function>
```

where the term *<function>* is replaced by the name of the function you are interested in. Check the text displayed to see if the directory directly after the word `toolbox` is **matlab** (which means it is core) or something else (which means it is from a toolbox).

For example:

```
>> which median
```

```
C:\Matlab\R2017b\Pro\toolbox\matlab\datafun\median.m
```

This shows us that the `median` function is part of the standard core Matlab distribution (notice how the word *matlab* follows the word `toolbox`). You may use the `median` function if you wish to.

```
>> which imadd
```

```
C:\Matlab\R2017b\Pro\toolbox\images\images\imadd.m
```

This shows us that the `imadd` function is part of the image processing toolbox and is therefore not permitted to be used for the project (notice how the word **images** follows the word `toolbox`).

## Summary of Functions to Write

There are **ten** in total:

| Function        | Difficulty Level | Where used  |
|-----------------|------------------|---|
| AlterByOne      | Easy             | Likely to be handy when writing your EmbedImage function  |
| ImageComplement | Easy             | Likely to be handy when writing your EncryptImage function. May also be useful when writing CreatePatterns. |
| CreatePatterns  | Easy             | CreateKey script  |
| GenerateKey     | Easy/Medium      | CreateKey script  |
| PatternsToImage | Medium           | CreateKey and Encryption scripts  |
| EmbedImage      | Medium/Hard      | CreateKey and Encryption scripts  |
| ImageToPatterns | Medium           | Encryption script   |
| EncryptImage    | Hard             | Encryption script   |
| ExtractImage    | Medium/Hard      | Decryption script   |
| DecryptImage    | Medium           | Decryption script   |

The pages following have detailed specifications of what each function should do. Remember that while the functions are designed to work together with the supplied scripts, many of the functions can be written and tested in isolation. Even if you get stuck on one function you should still try and write the others.

Keep the following general principles in mind:

- Note the capital letters used in function names. Case matters in MATLAB and you should take care that your function names EXACTLY match those in the projection specification. E.g. if the function name is specified as `AlterByOne` don't use the name `alterbyone`, `alterByOne`, or `AlterBy1`.
- The order type and dimension of input arguments matters. Make sure you have the required number of inputs in the correct order. E.g. if a function requires 2 inputs, as `DecryptImage` does (where the first input is a 2D array of uint8 values representing a cipher image and the second input is a 2D array of uint8 values representing a key image) then the order matters (i.e. the first input should be cipher image, to match the specifications). The number of inputs also matters (so don't change a function that requires 2 inputs to require more inputs).
- The type and dimensions of the outputs matter, e.g. `EmbedImage` returns a 3D array of uint8 colour values (representing an RGB colour image). Returning an array of doubles would cause your code to fail.

## The AlterByOne function

|          |   |
|----------|---|
| Purpose  | Adds 1 to an uint8 value in the range 0 to 255 inclusive (unless the value is 255 in which case it subtracts 1) |
| Input(s) | An integer value somewhere between 0 and 255 inclusive  |
| Output   | A uint8 value that is one more (unless the original value was 255, in which case it is one less)                |

This is one of the simplest functions to write. You may assume that this function will be passed a uint8 value and hence you can also assume that all values passed to it will be within the range 0 to 255 inclusive.

Your function should return 1 more than the value passed in, unless the value passed in was 255, in which case it should return 1 less (i.e. 254).

Hint: take care to make sure the value you return is of type `uint8`.

### Example calls

Here are some examples of calls to `AlterByOne`

```
>> v = AlterByOne(0)

v =

     1

>> v = AlterByOne(10)

v =

    11

>> v = AlterByOne(128)

v =

   129

>> v = AlterByOne(255)

v =

   254
```

## The ImageComplement function

|          |  |
|----------|--|
| Purpose  | Takes an image and swaps black for white and white for black   |
| Input(s) | A 2D array of uint8 values (i.e. a greyscale image) containing black and white pixels  |
| Output   | A 2D array of uint8 values (i.e. a greyscale image) of the complement of the input image (i.e. black has been swapped for white and white for black) |

This is a straight-forward function to write. It only needs to work for grayscale images (i.e. 2D arrays of uint8 values), it will not be required to handle colour images.

Remember that pure black is encoded as 0 and pure white is encoded as 255. Sometimes pixels are not quite pure black or pure white (but they are close). To the human eye they will still appear black and white, so we want our code to be robust enough to handle these values, i.e. we want to be able to swap dark pixels for light pixels (and vice versa).

To do this you should use the following formula to calculate the complement for each pixel in your image:

$$\text{Pixel Complement value} = 255 - \text{original pixel value}$$

This will map a pixel value of 255 to 0 and a pixel value of 0 to 255

It also has the advantage that it will handle “almost black” and “almost white” pixels.

e.g. 254 (very close to white) will map to  $255 - 254 = 1$  (very close to black)

### Example calls

Here are some examples of calls to ImageComplement

```
>> blackRectangle = zeros(2,3,'uint8')

blackRectangle =

    2x3 uint8 matrix

     0     0     0
     0     0     0

>> whiteRectangle = ImageComplement(blackRectangle)

whiteRectangle =

    2x3 uint8 matrix

    255    255    255
    255    255    255

>> blackAgain = ImageComplement(whiteRectangle)

blackAgain =

    2x3 uint8 matrix

     0     0     0
     0     0     0
```

```
>> p = uint8([254 255; 0 1])

p =

    2x2 uint8 matrix

    254    255
         0         1

>> c = ImageComplement(p)

c =

    2x2 uint8 matrix

         1         0
    255    254

>> rng(0) % set the random number generator seed
>> r = uint8(255*rand(3,4))

r =

    3x4 uint8 matrix

    208    233     71    246
    231    161    139     40
     32     25    244    248

>> c = ImageComplement(r)

c =

    3x4 uint8 matrix

     47     22    184         9
     24     94    116    215
    223    230     11         7
```

## The CreatePatterns function

|          |   |
|----------|---|
| Purpose  | Creates a 1D <b>cell</b> array containing six special 2x2 patterns of uint8 values, these patterns will be used for key generation  |
| Input(s) | None  |
| Output   | A single 1 x 6 <b>cell</b> array of patterns. It will contain the following six 2x2 patterns (in this order): bottom 2 black, top 2 black, left 2 black, right 2 black, leading diagonal black, off diagonal black<br>Each pattern will be stored as a 2x2 uint8 value array (i.e. a greyscale image) |

Here is a summary of the patterns to return (in order):

Pattern 1



Pattern 2



Pattern 3



Pattern 4



Pattern 5



Pattern 6



Pattern 1 (bottom 2 pixels black) will be stored in the first element of the cell array returned by the `CreatePatterns` function, whereas Pattern 3 (left 2 pixels black) will be stored in the third element of the cell array returned. You might find the `ImageComplement` function handy when writing `CreatePatterns` (although it isn't a requirement to use it).

### Example calls

Here is an example of a call to `CreatePatterns`

```
>> p = CreatePatterns()

p =
    1x6 cell array

    Columns 1 through 5
        {2x2 uint8}    {2x2 uint8}    {2x2 uint8}    {2x2 uint8}    {2x2 uint8}
    Column 6
        {2x2 uint8}

>> pattern1 = p{1} % note curly braces
pattern1 =
    2x2 uint8 matrix

    255    255
     0     0

>> pattern3 = p{3} % note curly braces
pattern3 =
    2x2 uint8 matrix

     0    255
     0    255
```

## The GenerateKey function

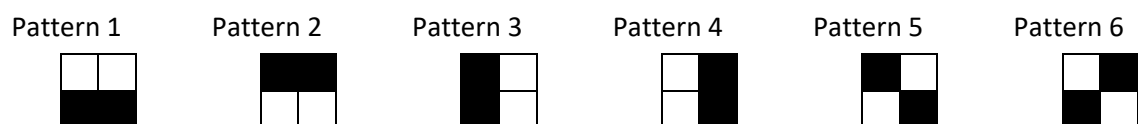
|          |  |
|----------|--|
| Purpose  | Assembles a 2D cell array to act as a key, where each array element is a 2x2 uint8 pattern selected at random. The patterns are passed in using a 1D cell array and then randomly assigned to the 2D cell array based on values in a 2D array of random values (that range from 1 to 6 inclusive)                              |
| Input(s) | Two inputs in the following order<br>1) A 2D m x n array of random integer values between 1 and 6 inclusive<br>2) A 1x6 1D cell array containing 6 patterns (where each pattern is stored as a 2x2 uint8 array)  |
| Output   | A 2D m x n <b>cell</b> array containing patterns to act as a key. Each element of the array will be a pattern stored as a 2x2 array of uint8 values (i.e. a grayscale image), with the pattern selected from the list of patterns based on the corresponding random values contained in the 2D array of random integer values. |

Suppose the first input was the array

r =

|   |   |   |   |
|---|---|---|---|
| 3 | 5 | 1 | 3 |
| 1 | 6 | 5 | 6 |
| 5 | 4 | 2 | 3 |

While the second input was a 1x6 1D cell array containing the following patterns



The output would be a 3x4 cell array containing the following patterns:

|        | Column 1 | Column 2 | Column 3 | Column4 |
|--------|----------|----------|----------|---------|
| Row 1: |          |          |          |         |
| Row 2: |          |          |          |         |
| Row 3: |          |          |          |         |



**Example calls**

Here is an example of working with `GenerateKey` (which assumes you have written the `CreatePatterns` function)

```
>> r = [3 5 1 3; 1 6 5 6; 5 4 2 3];

>> p = CreatePatterns();

>> key = GenerateKey(r,p)

key =

    3x4 cell array

    {2x2 uint8}    {2x2 uint8}    {2x2 uint8}    {2x2 uint8}
    {2x2 uint8}    {2x2 uint8}    {2x2 uint8}    {2x2 uint8}
    {2x2 uint8}    {2x2 uint8}    {2x2 uint8}    {2x2 uint8}

>> key{1,1} % note curly braces

ans =

    2x2 uint8 matrix

     0    255
     0    255

>> key{2,3} % note curly braces

ans =













    2x2 uint8 matrix

     0    255
    255     0
```

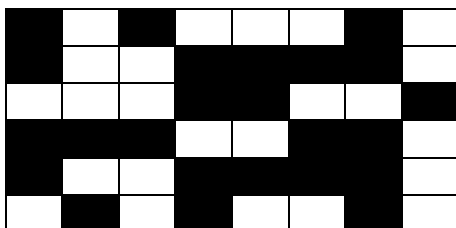
## The PatternsToImage function

|          |   |
|----------|---|
| Purpose  | Takes a cell array of 2x2 uint8 patterns (e.g. the patterns for our image) and converts it to a grayscale image (a 2D uint8 array) so that it can be displayed and saved  |
| Input(s) | A 2D m x n <b>cell</b> array of 2x2 patterns. Each element of the array will be a pattern stored as a 2x2 array of uint8 values (i.e. a grayscale image)  |
| Output   | A 2D array of uint8 values (i.e. a grayscale image) with the pixels assembled into a single image. This array will have twice the number of rows and twice the number of columns as the input array (i.e. it will be a 2m x 2n array) |

If the input to `PatternsToImage` was the 3x4 cell array containing the following patterns:

|        | Column 1   | Column 2   | Column 3   | Column 4   |
|--------|--|--|--|--|
| Row 1: |   |   |   |   |
| Row 2: |   |   |   |   |
| Row 3: |  |  |  |  |

The output would then be a 2D array of uint8 values that has this image:



(Remember black is represented as 0 and white as 255).

### Example calls

```
>> patternArrayRow = { uint8([ 0 0; 255 255]), uint8([0 255; 255 0]) }
patternArrayRow =
    1x2 cell array
        {2x2 uint8}    {2x2 uint8}
>> im = PatternsToImage(patternArrayRow)
im =
    2x4 uint8 matrix
         0         0         0    255
    255    255    255         0
```

The same example but this time using a semicolon (highlighted in red) when creating the cell array of patterns (so we have a column array)

```
>> patternArrayCol = { uint8([ 0 0; 255 255]); uint8([0 255; 255 0])}
patternArrayCol =
    2x1 cell array
    {2x2 uint8}
    {2x2 uint8}
>> im = PatternsToImage(patternArrayCol)
im =
    4x2 uint8 matrix
    0      0
    255    255
    0      255
    255     0
```

Here is an example of working with PatternsToImage (which assumes you have written the CreatePatterns and GenerateKey function)

```
>> r = [3 5 1 3; 1 6 5 6; 5 4 2 3];
>> p = CreatePatterns();
>> key = GenerateKey(r,p)
key =
    3x4 cell array
    {2x2 uint8}    {2x2 uint8}    {2x2 uint8}    {2x2 uint8}
    {2x2 uint8}    {2x2 uint8}    {2x2 uint8}    {2x2 uint8}
    {2x2 uint8}    {2x2 uint8}    {2x2 uint8}    {2x2 uint8}
>> keyImage = PatternsToImage(key)
keyImage =
    6x8 uint8 matrix
    0    255     0    255    255    255     0    255
    0    255    255     0     0     0     0    255
    255    255    255     0     0    255    255     0
    0     0     0    255    255     0     0    255
    0    255    255     0     0     0     0    255
    255     0    255     0    255    255     0    255
```

## The EmbedImage function

|          |  |
|----------|--|
| Purpose  | Used to embed a binary (i.e. black and white) image (stored as a grayscale image) inside another colour image (i.e. a 3D array of uint8 values representing an RGB image)  |
| Input(s) | Two inputs in the following order <ol style="list-style-type: none"> <li>1) A 2D array of uint8 values (i.e. a grayscale image) of the black and white image to embed inside the colour image. Pixel values will either be 0 or 255.</li> <li>2) A 3D array of uint8 values (i.e. an RGB colour image) that the black and white image will be embedded into</li> </ol> |
| Output   | A 3D array of uint8 values (i.e. an RGB colour image) that contains a hidden black and white image   |

To embed the binary (i.e black and white) image inside the colour image we use the following idea.

If a pixel in the binary image is **black** then we make sure that the red, green and blue values for the corresponding pixel in the colour image sum to an **even** number. If the RGB values do not already sum to an even number, we alter the red value by 1, to ensure an even sum.

If a pixel in the binary image is **white** then we make sure that the red, green and blue values for the corresponding pixel in the colour image sum to an **odd** number. If the RGB values do not already sum to an odd number, we alter the red value by 1, to ensure an odd sum.

When altering a red value by 1 we add 1 to the red value, unless it is already 255, in which case we subtract 1. You will likely find the `AlterByOne` helper function useful to call (although you are not required to use it).

Suppose the pixel in the first row, third column of our binary image is black (i.e. 0). We will encode this pixel by ensuring that the RGB values for the pixel in the first row, third column of our colour image sum to an even number. Say the RGB values for this pixel are R=3, G=10, B=100. The sum of these three colour values is  $3+10+100=113$ , which is an odd number, so we would need to alter the red value to ensure an even sum, hence we adjust the red value to be  $3+1=4$ .

The advantage of the above approach is that we can embed the entire image within the colour image with only very tiny changes to the amount of red in the image. These changes won't be noticeable to the human eye, so the resulting image will look identical to the original image but now hides extra information within it.

The binary image to embed needs to be the same size or smaller than the colour image. Your code should be able to handle embedding a binary image that is the same size as the colour image. It should also be able to handle embedding a smaller binary image within a large colour image (i.e. you should not assume that the two images are necessarily the same size). If the binary image is smaller, it will be embedded in the top left of the colour image.

You may assume that the binary image won't be bigger than the colour image (i.e. you don't need to do any error checking on the size of the images).

**Hint:** to check if a value is even, you can use the `mod` function, `mod(a, 2)` will return 0 if `a` is even and 1 if `a` is odd (as `mod(a, 2)` returns the remainder after division of `a` by 2).

### Example calls

Here is an example of working the with EmbedImage function at the command line

```
>> binary = imread('key.png');  
>> colour = imread('peter.png');  
>> hidden = EmbedImage(binary,colour);  
>> imwrite(hidden, 'hidden.png');
```

Here is another example working with some tiny 2x2 images (Take note of how the red values have changed)

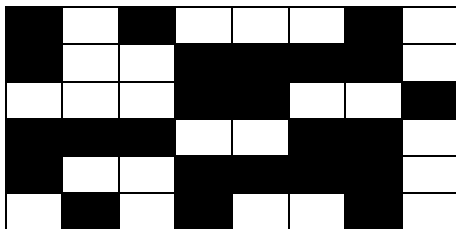
```
>> binary = uint8([ 0 255; 255 0])  
  
binary =  
  
    2x2 uint8 matrix  
  
      0      255  
    255       0  
  
>> rng(0) % set random seed  
>> colour = uint8(255*rand(2,2,3)) % you will get different values each time  
  
    2x2x3 uint8 array  
  
colour(:,:,1) =  
  
    208     32  
    231    233  
  
colour(:,:,2) =  
  
    161     71  
     25    139  
  
colour(:,:,3) =  
  
    244     40  
    246    248  
  
>> hidden = EmbedImage(binary,colour)  
  
    2x2x3 uint8 array  
  
hidden(:,:,1) =  
  
    209     32  
    232    233  
  
hidden(:,:,2) =  
  
    161     71  
     25    139  
  
hidden(:,:,3) =  
  
    244     40  
    246    248
```

## The ImageToPatterns function

|          |   |
|----------|---|
| Purpose  | Convert an image back into a cell array of 2x2 uint8 patterns so that we can more easily work with the patterns   |
| Input(s) | A 2D array of uint8 values (i.e. a grayscale image) containing an image made up of lots of black and white pixels   |
| Output   | A 2D <b>cell</b> array of 2x2 patterns extracted from the image. Each element of the array will be a pattern stored as a 2x2 array of uint8 values (i.e. a grayscale image) |

Note that this function is the inverse of `PatternsToImage` (i.e. it does the reverse process).

If the input was a 6x8 2D array of uint8 values that has this image:



Then the output of `ImageToPatterns` would be the 3x4 cell array containing the following patterns:

|        | Column 1 | Column 2 | Column 3 | Column 4 |
|--------|----------|----------|----------|----------|
| Row 1: |          |          |          |          |
| Row 2: |          |          |          |          |
| Row 3: |          |          |          |          |

### Example calls

Overleaf are some examples of calls to `ImageToPatterns`

Note that if you have already written `PatternsToImage` you can use that to help you test your function, as if both are written correctly then they should act as inverses of each other. This means that a call to

```
ImageToPatterns ( PatternsToImage ( p ) )
```

should just give you back your cell array of patterns, `p` and similarly

```
PatternsToImage ( ImageToPatterns ( im ) )
```

Should just give you back your image `im`.

```

>> im = uint8([0 0 0 255; 255 255 255 0])

im =

    2x4 uint8 matrix

      0      0      0    255
    255    255    255      0

>> p = ImageToPatterns(im)

p =

    1x2 cell array

    {2x2 uint8}    {2x2 uint8}

>> p1 = p{1} % note curly braces

p1 =

    2x2 uint8 matrix

      0      0
    255    255

>> p2 = p{2} % note curly braces

p2 =

    2x2 uint8 matrix

      0    255
    255      0

>> image = uint8([      0    255      0    255    255    255      0    255;
      0    255    255      0      0      0      0    255;
    255    255    255      0      0    255    255      0;
      0      0      0    255    255      0      0    255;
      0    255    255      0      0      0      0    255;
    255      0    255      0    255    255      0    255; ]);

>> p = ImageToPatterns(image)

p =

    3x4 cell array

    {2x2 uint8}    {2x2 uint8}    {2x2 uint8}    {2x2 uint8}
    {2x2 uint8}    {2x2 uint8}    {2x2 uint8}    {2x2 uint8}
    {2x2 uint8}    {2x2 uint8}    {2x2 uint8}    {2x2 uint8}

>> p13 = p{1,3}

p13 =

    2x2 uint8 matrix

    255    255
      0      0

```



## The EncryptImage function

|          |  |
|----------|--|
| Purpose  | Encrypt a plain image using a key array of patterns (a cell array containing 2x2 patterns), to obtain an encrypted cipher array of patterns (a cell array containing 2x2 patterns)   |
| Input(s) | Two inputs in the following order <ol style="list-style-type: none"> <li>1) A 2D array of uint8 values (i.e. a grayscale image) containing the black and white plain image to encrypt.</li> <li>2) A 2D <b>cell</b> array of 2x2 patterns to act as a <b>key</b> array of patterns. Each element of the cell array will be a pattern stored as a 2x2 array of uint8 values (i.e. a grayscale image)</li> </ol> |
| Output   | A 2D <b>cell</b> array of 2x2 patterns which is our <b>cipher array of patterns</b> (i.e. it was created by encrypting our plain image). Each element of the array will be a pattern stored as a 2x2 array of uint8 values (i.e. a grayscale image)  |

This is the heart of the project, where we encrypt an image using a key array of patterns. Note that the key array of patterns is not an image, rather it is a 2D cell array of patterns (for our purpose this will be a little easier to work with than working directly with a 2D array of uint8 values). Each element of the key array of patterns is a small 2x2 pattern stored as a 2D array of uint8 values (i.e. the key array of patterns is a cell array that contains lots of small 2x2 images).

Our plain image should be a binary image (i.e. black and white, with no shades of grey). An ideal binary image has pixels of only two values: black (i.e. 0) and white (i.e. 255). Depending on how you create your binary image it may not have values which are **perfectly** black and white (i.e. some “black” values might be close to 0 but not quite zero and some “white” values might be close to 255 but not quite 255). To deal with this problem we will define **dark** pixels to be those which have a value less than 128 (and light pixels to be those that are 128 or more).

To encrypt our plain image we use our key array of patterns and a very simple rule:

If a pixel from our plain image is dark (e.g. black):

Assign to the cipher array of patterns the **complement** of the pattern from the corresponding position in the key array of patterns

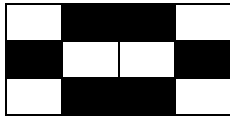
Otherwise:

Assign to the cipher array of patterns the **original** pattern from the corresponding position in the key array of patterns

You will notice that as we will often be assigning the complement a pattern, you may wish to use the helper function `ImageComplement` when doing this (although you are not required to do so, it will likely make your job easier!)

To better understand how encryption works let’s work through an example of how to encrypt a very small image with a very small key.

Suppose we have the tiny image below (a letter o):



Now suppose we have the following key array of patterns

|        | Column 1 | Column 2 | Column 3 | Column4 |
|--------|----------|----------|----------|---------|
| Row 1: |          |          |          |         |
| Row 2: |          |          |          |         |
| Row 3: |          |          |          |         |

To encrypt the pixel in row 1, column 1 of the image we will be using the pattern in row 1, column 1 of the key array of patterns. That is:



This pattern (or its complement) will be assigned to row 1, column 1 of the cipher array of patterns. Our plain image contains a white pixel in row 1, column 1, so we will be assigning the **original** pattern to row 1, column 1 of the cipher image.

To encrypt the pixel in row 1, column 2 of the image we will be using the pattern in row 1, column 2 of the key array of patterns. That is:



This pattern (or its complement) will be assigned to row 1, column 2 of the cipher array of patterns. Our plain image contains a black pixel in row 1, column 2, so we will be assigning the **complement** of this pattern to row 1, column 2 of the cipher image.

Repeating this process for all pixels in our plain image will produce the following cipher array of patterns:

|        | Column 1 | Column 2 | Column 3 | Column4 |
|--------|----------|----------|----------|---------|
| Row 1: |          |          |          |         |
| Row 2: |          |          |          |         |
| Row 3: |          |          |          |         |

**Example calls**

Here is an example of a call to `EncryptImage` (note that this relies on you having written the `ImageToPatterns` function)

```
>> tiny0 = uint8([255 0 0 255; 0 255 255 0; 255 0 0 255])

tiny0 =

    3×4 uint8 matrix

    255     0     0    255
     0    255    255     0
    255     0     0    255

>> keyImage = uint8([
        0    255     0    255    255    255     0    255;
        0    255    255     0     0     0     0    255;
       255    255    255     0     0    255    255     0;
        0     0     0    255    255     0     0    255;
        0    255    255     0     0     0     0    255;
       255     0    255     0    255    255     0    255; ]);

>> keyArray = ImageToPatterns(keyImage)

keyArray =

    3×4 cell array

    {2×2 uint8}    {2×2 uint8}    {2×2 uint8}    {2×2 uint8}
    {2×2 uint8}    {2×2 uint8}    {2×2 uint8}    {2×2 uint8}
    {2×2 uint8}    {2×2 uint8}    {2×2 uint8}    {2×2 uint8}

>> cipherArray = EncryptImage(tiny0,keyArray)

cipherArray =

    3×4 cell array

    {2×2 uint8}    {2×2 uint8}    {2×2 uint8}    {2×2 uint8}
    {2×2 uint8}    {2×2 uint8}    {2×2 uint8}    {2×2 uint8}
    {2×2 uint8}    {2×2 uint8}    {2×2 uint8}    {2×2 uint8}

>> c11 = cipherArray{1,1}

c11 =

    2×2 uint8 matrix

     0    255
     0    255

>> c12 = cipherArray{1,2}

c12 =

    2×2 uint8 matrix

    255     0
     0    255
```

## The ExtractImage function

|          |   |
|----------|---|
| Purpose  | Extract a black and white image that has been embedded within a colour image  |
| Input(s) | A 3D array of uint8 values (i.e. an RGB colour image) that contains a hidden black and white image  |
| Output   | A 2D array of uint8 values (i.e. a greyscale image) of the black and white image that was hidden inside the colour image (each pixel will have a value of 0 or 255) |

Note that this function is the inverse of `EmbedImage` function (i.e., it does the reverse process). We recover the hidden image by inspecting the value of the sum of red, green and blue values for each pixel in the colour image.

If the RGB values for a particular pixel sum to an **even** number then we assign a **black** pixel to the corresponding position in our binary image.

If the RGB values for a particular pixel sum to an **odd** number then we assign a **white** pixel to the corresponding position in our binary image.

Say the RGB values for the pixel in the first row, third colour of our colour image are R=4, G=10, B=100

The sum of these three colour values is  $4+10+100=114$ , which is an even number, so we would assign a black pixel (i.e. a value of 0) to the first row, third column of our binary image.

**Hint:** to check if a value is even, you can use the `mod` function, `mod(a, 2)` will return 0 if a is even and 1 if a is odd (as `mod(a, 2)` returns the remainder after division of a by 2).

### Example calls

Here is an example of a call to `ExtractImage` that relies on having written `EmbedImage` (showing that we should be able to get the original image back)

```
>> binary = uint8([ 0 255; 0 255])

binary =

    2x2 uint8 matrix

     0    255
     0    255

>> colour = uint8(255*rand(2,2,3)); % random values

>> hidden = EmbedImage(binary,colour)

>> original = ExtractImage(hidden)
original =

    2x2 uint8 matrix

     0    255
     0    255
```

Here is another example of a call to `ExtractImage` that doesn't rely on having written `EmbedImage` (as the hidden image is constructed manually).

```
>> hidden(:,:,1) = [178 242; 81 10];  
>> hidden(:,:,2) = [112 195; 97 203];  
>> hidden(:,:,3) = [48 114; 125 165];  
>> hidden = uint8(hidden) % make sure it is of type uint8  
  
2x2x3 uint8 array  
  
hidden(:,:,1) =  
  
    178    242  
     81     10  
  
hidden(:,:,2) =  
  
    112    195  
     97    203  
  
hidden(:,:,3) =  
  
     48    114  
    125    165  
  
>> m = ExtractImage(hidden)  
  
m =  
  
2x2 uint8 matrix  
  
     0    255  
    255     0
```

## The DecryptImage function

|          |   |
|----------|---|
| Purpose  | Decrypt a cipher image using the key image  |
| Input(s) | Two inputs in the following order <ol style="list-style-type: none"> <li>1) A 2D array of uint8 values (i.e. a grayscale image) containing the black and white <b>cipher</b> image (each pixel will have a value of 0 or 255)</li> <li>2) A 2D array of uint8 values (i.e. a grayscale image) containing the black and white <b>key</b> image (each pixel will have a value of 0 or 255)</li> </ol> |
| Output   | A 2D array of uint8 values (i.e. a grayscale image) containing the decrypted image, this will be the same size as the key.  |

Decrypting an image is much more straight forward than encrypting it. One reason is that we can work directly with the image arrays, rather than having to work with a 2D cell array of patterns.

To decrypt the image, we imagine the key and cipher images sitting one on top of the other and then examine each pair of corresponding pixels. (e.g. we compare the pixel from row 1, column 1 of the cipher with the pixel from row 1, column 1 of the key).

If two corresponding pixels are both white, then we add a white pixel to the corresponding position in our decrypted image.

What about if one or more of the corresponding pixels is black? We **could** assign any pair of corresponding pixels that has one or more black pixels the colour black. This would decode the image reasonably well, getting a result equivalent to laying a printed transparency of the key over top of a printed transparency of the cipher, where some areas of the image look black and others grey, however armed with a computer we can do **BETTER**, so will use a slightly different approach (which will result in a background that appears white rather than grey).

We know that if a pair of corresponding pixels contains TWO black pixels, that it is part of a pattern that can be thought of as sitting on top of itself (and hence not part of the original image), so we will add a white pixel to the corresponding position in our decrypted image.

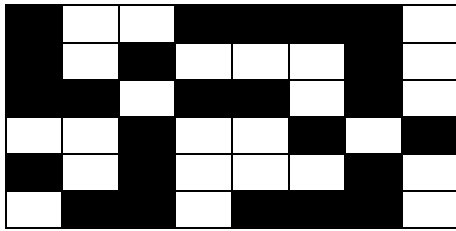
The only time we add a black pixel to the corresponding position in our decrypted image is when we have EXACTLY ONE black pixel in our little pair of corresponding pixels (it could be either the first or second pixel but there must be one black and one white). This happens when we have a pattern sitting on top of its complement.

If the above idea is applied to all the pixels that are “sitting on top of each other” the original image will be revealed, with some sections of the image appearing entirely black with other parts appearing white.

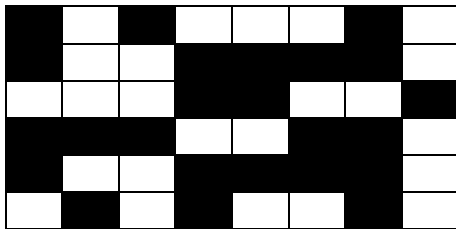
The key image needs to be the same size or smaller than the cipher image. You may assume that the key image won't be bigger than the cipher image.

Your code should be able to handle decrypting a cipher image that is the same size as the key image. It should also be able to handle decrypting part of a larger cipher image with a smaller key (in that case we line the images up so that the top left pixel of the key is sitting on the top left pixel of the cipher image and only examine the overlapping pixels, ignoring any pixels from the cipher image that don't have a corresponding pixel in the key image).

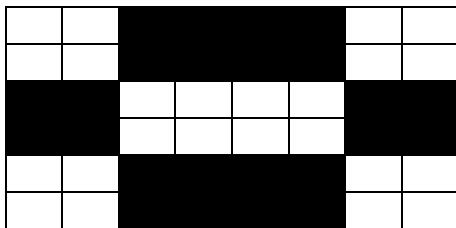
Suppose we have the following cipher image



Which has been encoded using the key below



Decrypting the image will yield



Note that even for this very small example image it is still possible make out the original message (it is a letter o) but in practice this method is applied to images that are several orders of magnitude larger than this example, which results in decrypted images which are easy to discern.

### Example calls

Here are some examples of calls to `DecryptImage`

```
>> cipherImage = uint8([ 0 255 255 0 0 0 0 255;
                          0 255 0 255 255 255 0 255]);

>> keyImage = uint8([ 0 255 0 255 255 255 0 255;
                      0 255 255 0 0 0 0 255]);

>> plainImage = DecryptImage(cipherImage,keyImage)

plainImage =

2x8 uint8 matrix
255 255 0 0 0 0 255 255
255 255 0 0 0 0 255 255
```



```
>> cipherImage = uint8([ 0 255 255 0 0 0 0 255;
                        0 255 0 255 255 255 0 255;
                        0 0 255 0 0 255 0 255;
                        255 255 0 255 255 0 255 0;
                        0 255 0 255 255 255 0 255;
                        255 0 0 255 0 0 0 255; ]);

>> keyImage = uint8([ 0 255 0 255 255 255 0 255;
                      0 255 255 0 0 0 0 255;
                      255 255 255 0 0 255 255 0;
                      0 0 0 255 255 0 0 255;
                      0 255 255 0 0 0 0 255;
                      255 0 255 0 255 255 0 255; ]);

>> plainImage = DecryptImage(cipherImage,keyImage)
plainImage =

6x8 uint8 matrix

    255    255     0     0     0     0    255    255
    255    255     0     0     0     0    255    255
     0     0    255    255    255    255     0     0
     0     0    255    255    255    255     0     0
    255    255     0     0     0     0    255    255
    255    255     0     0     0     0    255    255
```

## Checklist of Functions to Write in Alphabetical Order

Remember there are **ten** in total:

| Function        | Difficulty Level | Where used  |
|-----------------|------------------|---|
| AlterByOne      | Easy             | Likely to be handy when writing your EmbedImage function  |
| CreatePatterns  | Easy             | CreateKey script  |
| DecryptImage    | Medium           | Decryption script   |
| EmbedImage      | Medium/Hard      | CreateKey and Encryption scripts  |
| EncryptImage    | Hard             | Encryption script   |
| ExtractImage    | Medium/Hard      | Decryption script   |
| GenerateKey     | Easy/Medium      | CreateKey script  |
| ImageComplement | Easy             | Likely to be handy when writing your EncryptImage function. May also be useful when writing CreatePatterns. |
| ImageToPatterns | Medium           | Encryption script   |
| PatternsToImage | Medium           | CreateKey and Encryption scripts  |

## How the project is marked

You will receive marks both for style (is it well written?) and functionality (does your code work?) When you submit your code to MATLAB Grader it will run some tests to determine the marks for functionality. You can submit **each** function up to **six** times and your functionality mark for that function will be based on the function submission that performed the best. You will be marked for style by a human marker, who will download your function submissions (choosing the ones that performed the best) and mark these for style.

Each of the **ten** required functions will be marked independently for functionality, so even if you can't get everything to work, please do submit the functions you have written. Some functions may be harder to write than others, so if you are having difficulty writing a function you might like to try working on a different function for a while and then come back to the harder one later.

Note it is still possible to get marks for good style, even if your code does not work at all! Style includes elements such as using sensible variable names, having header comments, commenting the rest of your code, avoiding code repetition, and using correct indentation.

Each function can be written in less than 20 lines of code (not including comments) and some functions can be written using just a few lines (as an example my longest function was 16 lines of code and my shortest two were just 3 lines of code each, with an average length of 9 lines per function). Do not stress if your code is longer. It is perfectly fine if your project solution runs to many lines of code, as long as your code works and uses good programming style.

## How the project is submitted

Submission is done by uploading your code to MATLAB Grader. The submission area will be opened towards the end of week 6 in case you wish to submit early. You should be able to work on your project from anywhere, assuming you have access to MATLAB (either installed on a computer or running online) and can access the internet to upload your final submission.

Just as for the labs you will submit your project code by copying and pasting it into MATLAB Grader.

Before clicking the “Submit” button we **STRONGLY** recommend you try clicking on the “Run function” button, to ensure you have copied and pasted everything over correctly (otherwise you risk wasting a submission attempt on a copy/paste error).

**IMPORTANT:** when submitting a particular function, if your function calls helper functions (other than `AlterByOne` and `ImageComplement`) then you will need to paste the code for those functions below the particular function you are submitting, so that MATLAB grader has access to it.

## Any questions?

If you have any questions regarding the project, please first check through this document. If it does not answer your question, then feel free to ask the question on the class Piazza forum. Remember that you should **NOT** be publicly posting any of your project code on Piazza, so if your question requires that you include some of your code, make sure that it is posted as a **PRIVATE** piazza query.

Have fun!