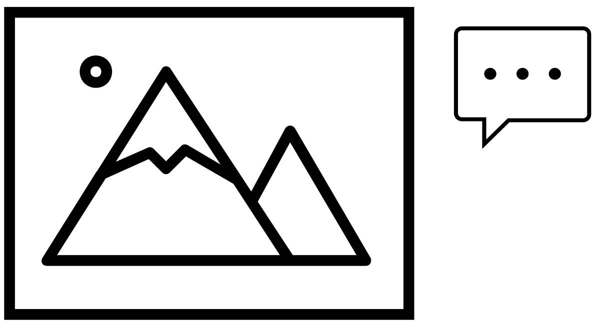
A Picture is Worth 10,000 Words

hiding language inside images



a

Duncan Bakke

production

in association with

Peter Bier Pictures

and

Kevin Jia Studios

additional editing by

Ashton Matthee

Project due date: You will need to upload your code before 11:59 pm on Sunday the 23rd of January.

Introduction

In this project, you will write code that will encrypt and decrypt strings within the pixel values of an image. In this manner you could hide instructions, a poem, or even ASCII art, and only someone with the key could retrieve it.

Cryptography may seem like a niche application of programming, but E2E (end-to-end) encryption in messaging, secure image storage, and login database protection are all essential parts of our Very Online™ world.

Cryptographic algorithms consist of encrypting a message at its source and decrypting it at its destination.

Encryption

The process of encryption uses a pre-defined key to lock up a message (plain text) into a form that is unreadable without the use of the key (cipher text). This means that interceptors cannot understand the message without first acquiring the key.

Decryption

Decryption is the inverse process; using the key to decrypt the cipher text into plain text so that it is once again usable. Encryption without a corresponding decryption process is just data destruction.

Embedding

In addition to the encryption process, we can further protect our cipher text by embedding it into a different file. In the case of this project, an image.

IMPORTANT

The MATLAB Project is intended to be challenging for beginners. How long it will take is entirely variable; some students will complete it in under an hour, some may take over 60.

Allow time to finish this project to the best of your ability.

In Summer School, the timeline is shortened compared to a normal semester, but the reduced class load should mean you have more time to dedicate to this assignment, so it ought to even out. Please feel free to ask questions in-lab and on Piazza (but don't post your code).

Do not copy anyone else's code, and do not let others copy yours.

All code is compared using a code-plagiarism detection software package developed by an international group of university software departments. If you copy/are copied, both parties will be considered guilty of misconduct, and you will receive zero for this project. If you have been caught cheating previously, you will fail the course. You will be caught. Do not do it.

This project is easier than you think it is.

Don't be daunted by how long this document is; it's purely due to the depth of explanation of each individual function you must write. Start with an easy one, keep going, you've got this.

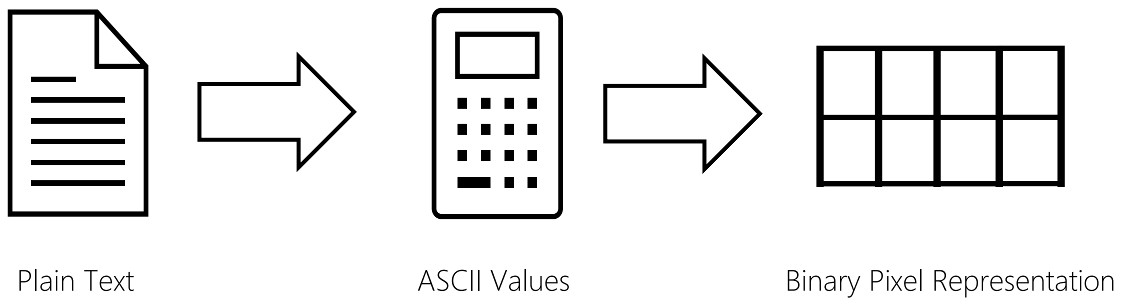
Encryption Steps

Here we will go through each of the steps that our programme will later perform, to help you understand the process and application of your code.

Step 1: Key Creation

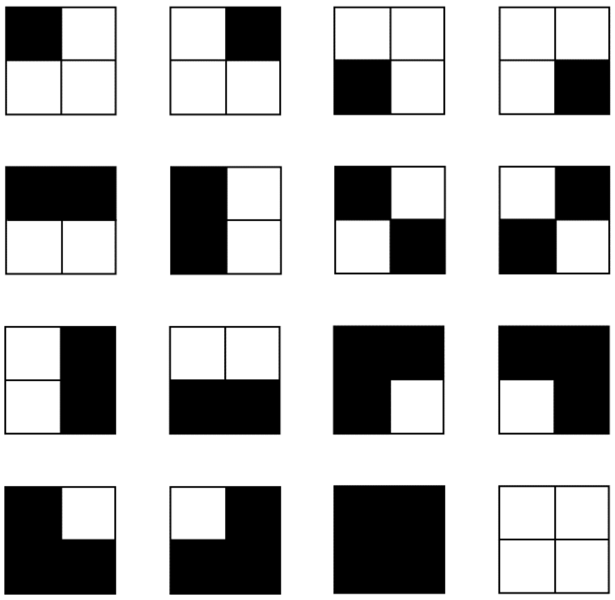
Before we can do anything else, we need to have a key. In this case, a key defines the representation of each character we want to be able to encode in the form of an 2x4 set of binary pixels (i.e. with values of only black and white, or 0 and 1).

Our key will include two components; one is widely known, the ASCII values for basic characters (*see Chapter 8: Strings*), and the second will be our own secret set of representations of those values.



Pixel Combinations

A 2x2 square set of black and white pixels can have 16 possible unique combinations, as below. This means that we could represent a series of numbers using these squares, so long as all of them were between 1 and 16 inclusive. To make this work for our 127 different ASCII values, we'll need to use two of these squares sequentially, so that we could represent any number between 1 and 16^2 (256). How we organise these squares becomes our unique key! There are `16!` (or 2.0923 x 10^{13}) possible orders for them to be in. So even if someone found our hidden blocks, it would take a long time to find the right strategy to decrypt them.



Hexadecimal Numbers

This concept isn't part of the curriculum, but it's a cool thing to know, and an important part of how this step works!

We normally use a decimal or base 10 number system (likely because we have ten fingers!), but that number is actually arbitrary. Other base systems work just as well, and some are very useful in certain situations. Binary, for example, uses a base 2 number system, since each bit can only have 2 possible values (0 or 1).

Since each of our 2x2 pixel squares has 16 possible values, we're going to convert ASCII numbers into Hexadecimal format before representing them in pixel blocks. That means that each digit can have 16 values, rather than 10! We use the first six letters of the alphabet to stand in for these extra values.

0 1 2 3 4 5 6 7 8 9 A B C D E F

It also means that two hexadecimal digits can represent 16^2 values (0-255), far more than the 10^2 values (0-99) that two decimal digits can handle.

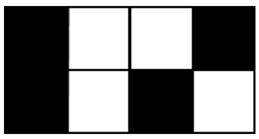
This is a hard idea to wrap your mind around, so don't worry too much if you don't get it at first; the **`dec2hex()`** and **`hex2dec()`** functions in MATLAB will do all the hard work for you. Note that dec2hex receives an integer value and returns the hex values in a *character array* (e.g. `dec2hex(229)` gives `'E5'`) and hex2dec receives a character array and returns the decimal value as a *value* (e.g. `hex2dec('E5')` gives `229`);

Step 2: Encryption

Our key can then be used to turn characters into blocks of 4x2 binary pixels.

In this example, the character of a capitalised w "W" is converted first into its ASCII value (87), then into the hexadecimal representation of that value ('57'), then the 5th and 7th combinations of the 2x2 squares are combined to form one 2x4 rectangle that represents "W". Likewise with "Hello".

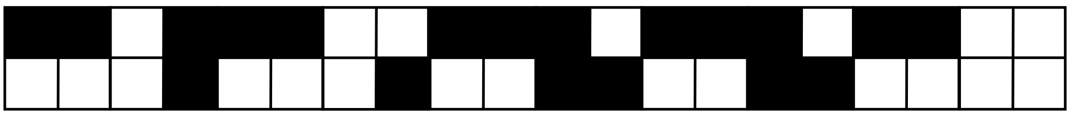
"W" DC: 87 | HX: '57'



Plain Text ASCII Values Binary Pixel Representation

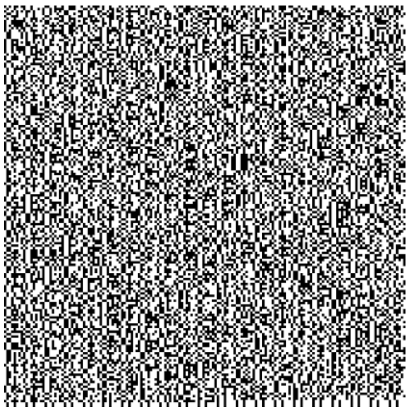
"Hello" '48' '65' '6C' '6C' '6F'

Plain Text ASCII Hex Values



Binary Pixel Representation

These blocks of pixels can then be combined into a viewable image of black and white pixels, by converting the 1 values into 255, and making all values of type `uint8` (*see Chapter 6*). In this project, we'll always be making square images. Since we know that to decode it, we just have to rearrange it into a 2 x (n×4) array of values, we can loop the row of pixels back on itself as many times as we like to make it square! Here's an example of 500 words of Latin text turned into a black and white encrypted image.



Note that not all text strings have a perfectly "squarable" length, but I'll only be testing your code with text that can be rearranged into a perfect square. Making your code handle strings that aren't "squarable" would be a great feature!

Step 3: Embedding

Now that we can turn text into blocks of pixels, here is where the magic happens; we hide those blocks of pixels inside otherwise normal images! Here's how.

Every pixel in our encrypted message is either a 1 (white) or 0 (black). We can hide this image inside another of greater size, in plain sight. The way we do this is by changing the RGB values of the image we are embedding it into so that the sum of all three is either odd (if the value of the pixel in our encrypted image is white) or even (if the value is black). That way, we can go back through and "find" our image by testing the sum of the RGB values.

How do we know which pixels to convert?

Note that as above, we can only hide our squares inside an image that is larger than they are. We will also only hide them starting from the upper left-hand corner. This gives us one more tool; we know that the bottom right pixel will never be part of the message! We can then use it to define the size of the square that we've hidden, by altering the sum of the RGB values.

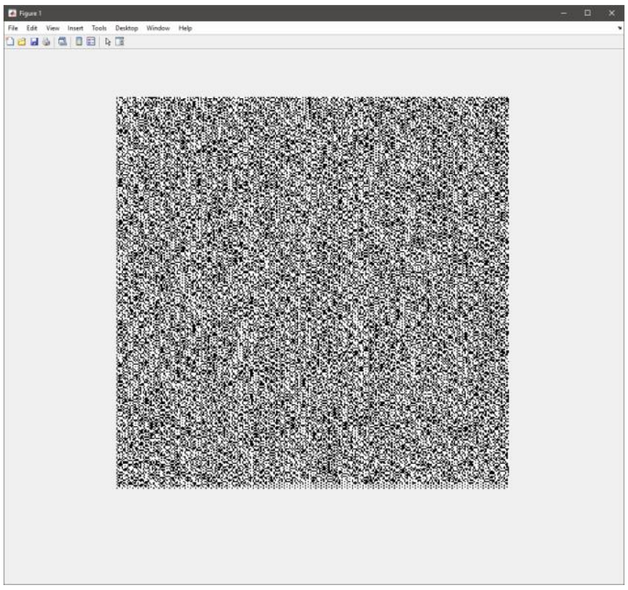
For example, the Latin square above is 168 pixels wide. If we were to embed it into an image, we'd also hide the number 168 in the bottom right pixel of the image we're embedding it into, by making it so that the sum of those RGB values is 168.

[Step 4: Extract and Decrypt]

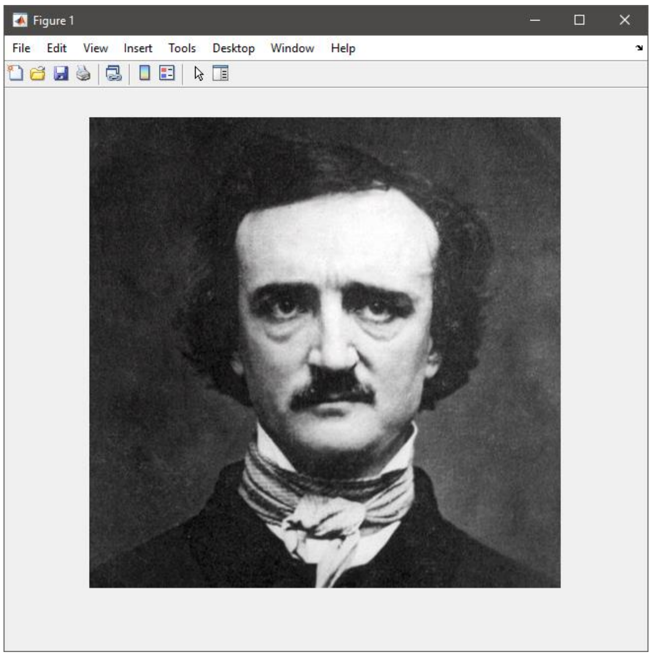
Then, with the right key, we should be able to extract and read the original message!

Example Encryption: The Tell Tale Heart

```
>> load tellTaleHeart %I've already stored the text in a variable
>> basePatterns = CreatePatterns();
>> patternKey = CreatePatternKey(basePatterns,secretIntegerArray);
>> tthImage = MakeCipherImage(tellTaleHeart,patternKey);
>> imshow(tthImage);
```



```
>> poeImage = imread('poe.jpg');
>> poesHiddenHeart = HideCipher(tthImage,poeImage);
>> imshow(poesHiddenHeart);
```



```
>> hisHideousHeart = FindCipher(poesHiddenHeart);

>> ReadCipherImage(hisHideousHeart,patternKey)

ans =

    'TRUE! NERVOUS VERY, very dreadfully nervous I had been and am;
but why will you say that I am mad? The disease had sharpened my
senses not destroyed not dulled them. Above all was the sense of
hearing acute. I heard all things in the heaven and in the earth. I
heard many things in hell. How, then, am I mad? Hearken! and observe
how healthily how calmly I can tell you the whole story. It is
impossible to say how first the idea entered my brain; but once
conceived, it haunted me day and night. Object there was none.
Passion there was none. I loved the old man. He had never wronged
me. He had never given me insult. For his gold I had no desire. I
think it was his eye! yes, it was this! He had the eye of a vulture
a pale blue eye, with a film over it. Whenever it fell upon me, my
blood ran cold; and so by degrees very gradually I made up my mind
to take the life of the old man, and thus rid myself of the eye
forever. Now this is the point. You fancy me mad. Madmen know
nothing. But you should have seen me. You should have seen how
wisely I proceeded with what caution with what foresight with what
dissimulation I went to work! I was never kinder to the old man than
during the whole week before I killed him. And every night, about
midnight, I turned the latch of his door and opened it oh so gently!
And then, when I had made an opening sufficient for my head, I put
in a dark lantern, all closed, closed, that no light shone out, and
then I thrust in my head. Oh, you would have laughed to see how
cunningly I thrust it in! I moved it slowly very, very slowly, so
that I might not disturb the old man's sleep. It took me an hour to
place my whole head within the opening so far that I could see him
as he lay upon his bed. Ha! would a madman have been so wise as
this, And then, when my head was well in the room, I undid the
lantern cautiously oh, so cautiously cautiously (for the hinges
creaked) I undid it just so much that a single thin ray fell upon
the vulture eye. And this I did for seven long nights every night
just at midnight but I found the eye always closed; and so it was
impossible to do the work; for it was not the old man who vexed me,
but his Evil Eye. And every morning, when the day broke, I went
boldly into the chamber, and spoke courageously to him, calling him
by name in a hearty tone, and inquiring how he has passed the night.
So you see he would have been a very profound old man, indeed, to
suspect that every night, just at twelve, I looked in upon him while
he slept. Upon the eighth night I was more than usually cautious in
opening the door. A watch's minute hand moves more quickly than did
mine. Never before that night had I felt the extent of my own powers
of my sagacity. I could scarcely contain my feelings of triumph. To
```

FAQ

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 `HideCipher` and `FindCipher` 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 many 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

You will be provided with three scripts called `CreateKey.m`, `Encryption.m` and `Decryption.m`. Note that these scripts do *not* contain the same code as in semester 2 of 2021. These will call your functions, such that once you've written all of them successfully, they will perform their respective tasks (creating a key, encrypting text, decrypting an image).

There are eight functions for you to write.

Function	Difficulty	Where Used
`AlterByOne`	Easy	Useful for making your `HideCipher` function.
`CreatePatterns`	Easy	`CreateKey` script
`CreatePatternKey`	Medium	`CreateKey` script
`GetHexASCII`	Medium	Useful for making your `MakeCipherImage` function.
`MakeCipherImage`	Medium	`Encryption` script.
`HideCipher`	Hard	`Encryption` script.
`FindCipher`	Medium	`Decryption` script.
`ReadCipherImage`	Hard	`Decryption` script.

The rest of this document has 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, as all are worth marks individually. The rest of this document contains example calls so you can test your code, but you should write your own tests as well; not every case that we will test for in marking has been included.

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 `ReadCipherImage` 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 binary values representing a pattern key) 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.,

For example, `HideCipher` returns a 3D array of uint8 colour values (representing an RGB colour image). Returning an array of doubles would cause your code to fail.

AlterByOne

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`.

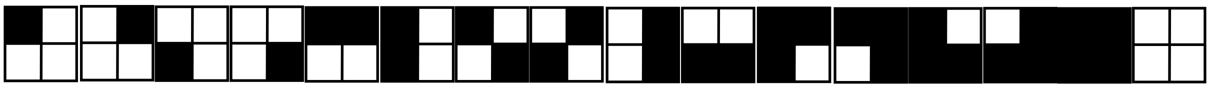
Example calls

```
>> v = AlterByOne(0)
 v =
      1

>> v = AlterByOne(10)
 v =
      11

>> v = AlterByOne(128)
 v =
      129

>> v = AlterByOne(255)
 v =
      254
```

Create Patterns

Purpose	Creates a 1D cell array containing 16 special 2x2 patterns of uint8 values, these patterns will be used for key generation
Input(s)	None
Output	A single 16 x 1 cell array of patterns. It will contain the following 16 2x2 patterns (in this order): top left black, top right black, bottom left black, bottom right black, top 2 black, left 2 black, leading diagonal black, off diagonal black, right 2 black, bottom 2 black, top 2 and bottom left black, top 2 and bottom right black, bottom 2 and top left black, bottom 2 and top right black, all black, all white. Each pattern will be stored as a 2x2 uint8 value array (i.e. a greyscale image)

The patterns you must store (in order from 1 – 16):



This is the same order as the depiction in the introduction, from left to right and top to bottom. This function will always return the same output.

Example call

```
>> patternArray = CreatePatterns();
patternArray =
  16×1 cell array
>> whos patternArray
  Name                 Size              Bytes  Class    Attributes
  patternArray        16x1                1728  cell
>> patternArray{8}
ans =
  2×2 uint8 matrix
   1   0
   0   1
>> patternArray{11}
ans =
  2×2 uint8 matrix
   0   0
   0   1
```

If you find any typos or mistakes, please let me know and I will fix them and reupload this document.

CreatePatternKey

Purpose	Arranges a 1D cell array containing 16 special 2x2 patterns of uint8 values based on both an existing 16 x 1 cell array of patterns and a 16 x 1 array of integers from 1-16 describing the order in which they should be arranged.
Input(s)	This function has two inputs (in this order): 1. A 1D 16 x 1 cell array of patterns 2. A 1D 16 x 1 array of integers from 1-16. (each integer will appear only once).
Output	A single 16 x 1 cell array of 2x2 patterns of uint8 values (should only be 0 and 1). These are the same patterns as the cell array input, but in a new order.

This function is what defines the key we'll use to encrypt our message; there are 2.0923 x 10^{13} possible combinations of the same 16 patterns, so the likelihood of someone stumbling upon the precise version that we use is minimal. Someone could attempt a smarter analysis by checking what blocks turn up lots (and then assuming those patterns are spaces, or the letter 'E', the most common letter in English), but a brute force decryption would be very difficult.

There are a few ways to go about rearranging a cell array; one thing to remember is not to overwrite the input! If you assign one cell element to another, the previous occupant of that element will be gone.

In the Example Calls that follow, you'll see the rearrangement concept demonstrated using the specific examples of an identical cell array (i.e. the integer array input was `[1:16]`) and an inverted cell array (i.e. the integer array input was `[16:-1:1]`). These are simply illustrations; the real integer arrays could be in any order at all!

Example Calls

```
>> patternArray = CreatePatterns();

patternArray =

  16×1 cell array

>> patternKeyIdentical = CreatePatternKey(patternArray,[1:16]);

patternKeyIdentical =

  16×1 cell array

>> patternKeyIdentical{1} == patternArray{1}

ans =

  2×2 logical array

   1   1

   1   1
```

<continued on next page>

```
>> patternKeyIdentical{15} == patternArray{16}
ans =
  2×2 logical array
   0   0
   0   0
>> patternKeyInverted = CreatePatternKey(patternArray,[16:-1:1]);
>> patternKeyInverted{1} == patternArray{1}
ans =
  2×2 logical array
   0   0
   0   0
>> patternKeyInverted{1} == patternArray{16}
ans =
  2×2 logical array
   1   1
   1   1
```

GetHexASCII

Purpose	Turns a character array input into a character array output where each element is the hexadecimal ASCII value of the corresponding input character.
Input(s)	One input, a 1D character array of any length *n*.
Output	A 2D (*n* x 2) character array where each row is two hexadecimal digits (stored as characters) representing the hexadecimal ASCII value of each input character.

This function is the first part of our encryption process. It converts our character array of text into hexadecimal values for our second encryption step. At the end of this function, each row of the character array output represents a single character of the input array.

Example Calls

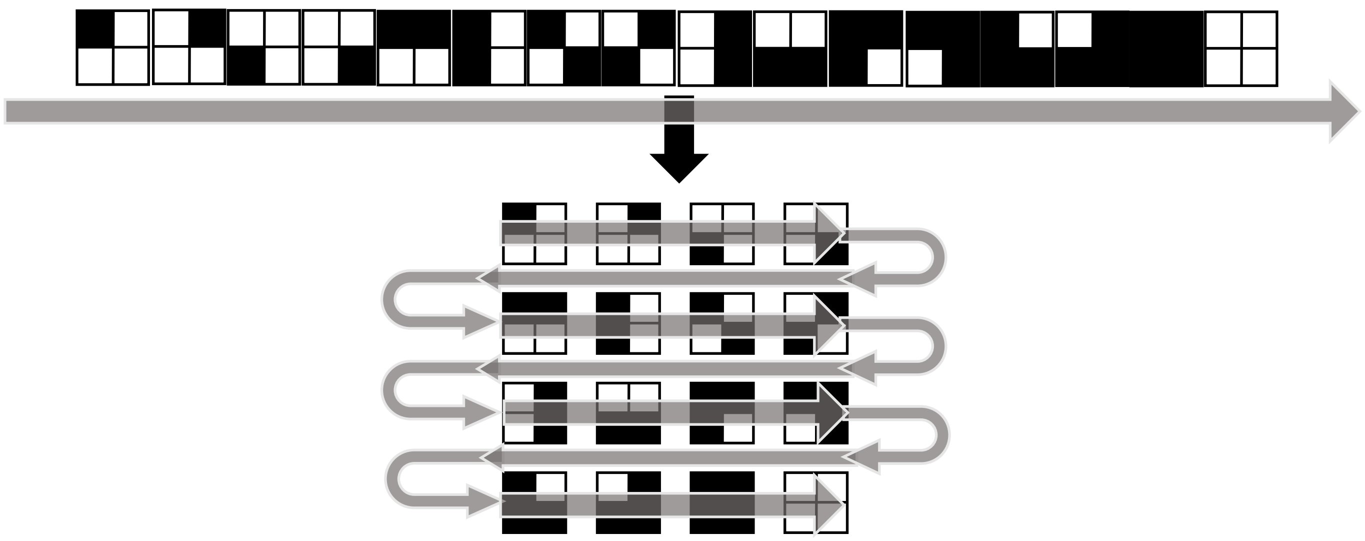
```
>> GetHexASCII('Hello')


ans =

  5×2 char array

    '48'

    '65'

    '6C'

    '6C'

    '6F'


>> GetHexASCII('Goodbye')


ans =

  7×2 char array

    '47'

    '6F'

    '6F'

    '64'

    '62'

    '79'

    '65'
```

MakeCipherImage

Purpose	Turns a 1D character array and a 1D cell array of patterns into a square black-and-white pattern containing an encrypted version of the 1D character array
Input(s)	Two inputs, in the following order: 1. a 1D character array of any length *n*. 2. a 2D cell array containing 16 2x2 uint8 binary patterns (created by CreatePatternKey)
Output	A 2D array of uint8 values depicting a black-and-white image made up of the 1D character array encrypted into 2x4 blocks of pixels (all pixels will have values of either 0 or 255).

This is a big function, the one that makes words into pictures. You'll likely find that calling `GetHexASCII` will be useful in this function. While it is not a trivial task to encode the characters into a 2 x (n*4) uint8 matrix full of their pixel block representations, you must also then reshape that matrix into a square. Remember two important things:

1. While you are reshaping, never break the top and bottom half of a pixel block apart! We can easily attach all the 2-pixel-wide rows back together, but only when we only have to worry about one of the two dimensions. I've done an example of this process below, with the pattern row from before (the arrows show the direction of the text inside).



2. While the index of the patterns is 1-16, you are representing digits with value *0 – 15*! Or, more accurately, 0 – F. Be sure you are using the right index to describe your characters.

Note: As previously mentioned, I will only be testing your code with character arrays that will be perfectly "squarable" (i.e. the number of characters will be such that they can be rearranged as above into an equal-sided square. However, you could make your code immune to non-squarable arrays by including a dynamically-sized buffer of spaces, or some other solution of the like. Feel free to experiment!

Example Calls

```
>> MakeCipherImage('Hi',keyPattern)


ans =


  4×4 uint8 matrix


     0     0   255     0

   255   255   255     0

     0   255   255   255

   255     0     0     0


>> MakeCipherImage('wassuuup',keyPattern)


ans =


  8×8 uint8 matrix


   255     0   255     0     0   255   255     0

     0   255     0   255   255     0   255   255

   255     0   255   255   255     0   255   255

     0   255   255     0     0   255   255     0

   255     0     0   255   255     0     0   255

     0   255     0   255     0   255     0   255

   255     0     0   255   255     0     0   255

     0   255     0   255     0   255   255   255
```

HideCipher

Purpose	Takes a square black and white image and hides it in a larger, colour image by altering the sums of the RGB values in that image. It also uses the bottom right pixel to store the size of the square, so that it can be later decoded.
Input(s)	Two inputs, in the following order: 1. a 2D array of uint8 values (only should have values of 0 and 255) to be hidden 2. a 3D array of uint8 values (RGB) in which to hide the first. First two dimensions must be larger than the square!
Output	A 3D array of uint8 values (RGB) that has the black and white image encoded into it, as well as the last pixel (bottom right) set so that its RGB values sum to the size of the hidden square.

To embed the binary 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 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.

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 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.

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 odd number. Say the RGB values for this pixel are R=3, G=10, B=101. The sum of these three colour values is 3+10+101=114, which is an even number, so we would need to alter the red value to ensure an odd 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 at least one pixel smaller than the image it is being embedded in, in at least one dimension. You can assume this will be the case (i.e. you do not need to perform error checking for image sizes).

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).

In addition to the above, we need a way to communicate to the recipient of this function the size of the hidden square beginning from the top left. We will do this by altering the sum of the RGB values in the bottom right hand corner to sum to the size of the square in any one dimension.

Suppose we are hiding the entire text of Edgar Allen Poe's "The Tell-Tale Heart" inside an image. After our encrypting (and me including some extra spaces to get it square), it becomes a square of pixels 300 wide. Suppose the bottom-right pixel in the image we are hiding it in is a very flat grey (RGB = [125, 125, 125]). We would then set those values to add to 300. We could do it in a variety of ways; we could turn it bright orange by setting the values to [255,45,0]. We could try and keep it inconspicuous by altering all three to [100, 100, 100]. You will not be marked on your stylistic choices; solve this problem however you wish, as long as these three numbers add to the size of your square!

Example Calls

```
>> binary = uint8([ 0 255; 255 0])
binary =
  2×2 uint8 matrix
     0   255
   255     0
>> colour = uint8(255*rand(2,2,3)) %these will be different
  2×2×3 uint8 array
colour(:,:,1) =
   181    70
   192   173
colour(:,:,2) =
   167    30
    41   127
colour(:,:,3) =
   245   149
    87    57
```

```
>> hidden = HideCipher(binary,colour)

  2×2×3 uint8 array

hidden(:,:,1) =

   181    71

   192   173

hidden(:,:,2) =

   167    30

    41   127

hidden(:,:,3) =

   245   149

    87    57

% Note how the red value for the upper right pixel changed
(because it originally summed to an odd number) but the lower
left pixel stayed the same (because it already summed to an even
number).
```

Here's how it might look if we were using it on images in the command window:

```
>> binary = imread('rickastley.png');

>> colour = imread('duncanneo.png');

>> rickroll = HideCipher(binary,colour);

>> imwrite(rickroll, 'totallynotarickroll.png';
```

FindCipher

Purpose	Takes a colour image and retrieves the hidden binary square inside it, by checking the bottom right pixel for size and the RGB sums for binary value.
Input(s)	One input: a 3D array of uint8 values (RGB) in which a binary square is hidden starting from the upper left corner.
Output	A 2D array of uint8 values (Black and White) retrieved from its hiding place.

This function is a bit easier once you have completed HideCipher, so I recommend you start there. This function performs the inverse (i.e. it uncovers the embedded cipher image, rather than placing it).

We first check the sum of the red, green, and blue values for the bottom right pixel; suppose it had the values [100, 100, 100]; We would then know that the binary square was hidden in pixels [1,1] through [300,300].

We recover the hidden image by inspecting the value of the sum of red, green, and blue values for each pixel in the range we've identified. If the RGB values for a particular pixel sum to an odd 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 even 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 white pixel (i.e. a value of 255) 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

```
>> hiCipher = MakeCipherImage('Hi',keyPattern);
hiCipher =
  4×4 uint8 matrix
     0     0   255     0
   255   255   255     0
     0   255   255   255
   255     0     0     0
>> hiddenHi = HideCipher(hiCipher,colourImage);
>> retrievedHiCipher = FindCipher(hiddenHi)
retrievedHiCipher =
  4×4 uint8 matrix
     0     0   255     0
   255   255   255     0
     0   255   255   255
   255     0     0     0

>> colour = uint8(255*rand(5,5,3)); %Always different
>> hiddenHi = HideCipher(hiCipher,colour)
  5×5×3 uint8 array
hiddenHi(:,:,1) =
   193   245   214    90    90
    66   141    66    50   212
   129    35   209    65   149
   178    38    62   158   140
   227    66   237   121     1
hiddenHi(:,:,2) =
    73    19    33    41   154
   193    14   145   203    67
   192   135   120    79   167
    97   199     3   135   176
   145   238    86    42     1
```

```
hiddenHi(:,:,3) =


   115   211    27   222   110

    21   137   245    22   232

    58   254     1   102    46

   233    20   198    66    67

    39   113   208   204     2


% Note the sums of the bolded elements; odd corresponds to black
cipher pixels, even to white cipher pixels, and the final pixel
now sums to 4, the size of the cipher square.


>> retrievedHiCipher = FindCipher(hiddenHi)

retrievedHiCipher =


  4×4 uint8 matrix


     0     0   255     0

   255   255   255     0

     0   255   255   255

   255     0     0     0
```

ReadCipherImage

Purpose	Retrieve the text that was encrypted into the binary image, using the same key as the one that encrypted it.
Input(s)	Two inputs, in the following order: 1. A 2D array of uint8 values depicting a square black-and-white image (all pixels will have values of either 0 or 255). 2. a 2D cell array containing 16 2x2 uint8 binary patterns (created by CreatePatternKey) (*Note: this must be the same one that was used to encrypt!)*
Output	A 1D character array containing the hidden message encrypted in the black and white image.

NOTE: You may find this function very hard.

It combines many ideas, including (but not limited to) array subsections, logical indexing, and character representations in decimal and hexadecimal forms. You may find Matlab concepts not covered in this course useful if you're aiming for highly-efficient code (`cellfun`). This function is intended to be a differentiator; the difference between an A and an A+ in the course.

All this to say that if you have a hard time with this function, don't worry; I did too.

Since we used the key to encrypt the text into the image, the image and the key are all we need to turn that image back into text. Here are some steps you could follow (though as always, if you come up with some cleverer solution, don't let me stop you!):

1. Rearrange the 2D array from a square into a 2 x *m* array (where *m* = *n* x 4, *n* being the original number of words in the image. How might you work out the value of *n*? How would we take each 2 row subsection and concatenate them horizontally?
2. In the 2 x *m* 2D array, each 2 x 4 block should represent one letter in hexadecimal, each 2 x 2 half being one hexadecimal digit from 0 – F. How can you turn the value of the index of the corresponding array in the key into a hexadecimal digit? How can you combine the digits and convert them back to decimal?
3. Once you have the decimal ASCII values, you need to turn them back into a character array. How would you do that?

Again, I realise this function can be a hard task for a new programmer; difficult tasks are an important part of learning and assessment. You can complete this solely with material taught in class.

Example Calls

```
>> cipherArray = MakeCipherImage('Hi',keyPattern)

cipherArray =

  4×4 uint8 matrix

      0      0    255      0

    255    255    255      0

      0    255    255    255

    255      0      0      0


>> ReadCipherImage(cipherArray,keyPattern)

ans =

    'Hi'


>> cipherArray = MakeCipherImage('wassuuup',keyPattern)

cipherArray =

  8×8 uint8 matrix

    255      0    255      0      0    255    255      0

      0    255      0    255    255      0    255    255

    255      0    255    255    255      0    255    255

      0    255    255      0      0    255    255      0

    255      0      0    255    255      0      0    255

      0    255      0    255      0    255      0    255

    255      0      0    255    255      0      0    255

      0    255      0    255      0    255    255    255


>> ReadCipherImage(cipherArray,keyPattern)

ans =

    'wassuuup'
```

Function Checklist

All the functions you must write are listed below in alphabetical order, to help you make sure you've completed everything before submitting.

	Function Name	Difficulty	Description
☐	`AlterByOne`	*Easy*	Adds 1 to an uint8 value in the range 0 to 255 inclusive (unless the value is 255 in which case it subtracts 1).
☐	`CreatePatternKey`	*Medium*	Arranges a 1D cell array containing 16 special 2x2 patterns of uint8 values based on both an existing 16 x 1 cell array of patterns and a 16 x 1 array of integers from 1-16 describing the order in which they should be arranged.
☐	`CreatePatterns`	*Easy*	Creates a 1D cell array containing 16 special 2x2 patterns of uint8 values, these patterns will be used for key generation.
☐	`FindCipher`	*Hard*	Takes a colour image and retrieves the hidden binary square inside it, by checking the bottom right pixel for size and the RGB sums for binary value.
☐	`GetHexASCII`	*Medium*	Turns a character array input into a character array output where each element is the hexadecimal ASCII value of the corresponding input character.
☐	`HideCipher`	*Hard*	Takes a square black and white image and hides it in a larger, colour image by altering the sums of the RGB values in that image. It also uses the bottom right pixel to store the size of the square, so that it can be later decoded.
☐	`MakeCipherImage`	*Medium*	Turns a 1D character array and a 1D cell array of patterns into a square black-and-white pattern containing an encrypted version of the 1D character array.
☐	`ReadCipherImage`	*Hard*	Retrieve the text that was encrypted into the binary image, using the same key as the one that encrypted it.

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 **eight** 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 19 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, much like the lab submissions. The submission area will be opened towards the end of Week 3 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 of running online) and can access the internet to upload your final submission.

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` or `GetHexASCII`) 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!