You must work either on your own or with one partner. If you work with a partner you must first register as a group in CMS and then submit your work as a group. Adhere to the Code of Academic Integrity. For a group, “you” below refers to “your group.” You may discuss background issues and general strategies with others, but the work that you submit must be your own. In particular, you may discuss general ideas with others but you may not work out the detailed solutions with others. It is not OK for you to see or hear another student’s code and it is certainly not OK to copy code from another person or from published/Internet sources. If you feel that you cannot complete the assignment on your own, seek help from the course staff.

Objectives

Completing this part of the project will solidify your understanding of 3-dimensional arrays and the types \texttt{uint8} and \texttt{char} through an image processing application. Pay attention to the difference between \texttt{uint8} and MATLAB’s default type \texttt{double}.

**Restriction:** Do not use built-in function \texttt{find} or other functions not discussed in the course.

2 Digital Steganography

Can you see the difference between the above images? I can’t, but in fact a message has been encrypted in the image on the right. Steganography is the art and science of hiding messages in such a way that no one, apart from the sender and receiver, suspects the existence of the message.\footnote{Wikipedia. http://en.wikipedia.org/wiki/Steganography} Generally, one can hide a message successfully in a large set of noisy data. A digital image is a good candidate medium for hiding messages because it comprises a large number of pixels and small changes to the individual RGB values are not discernible to the eye. You will write a set of functions to encrypt (hide) a message in a color image and decrypt (reveal) the message.

How do we hide a character, a message?

To hide an individual character of a message in a host image, we need to modify some RGB values in the image. Since image data is numeric (type \texttt{uint8}), a natural choice is to modify the RGB values by the ASCII code of the character. (Recall that every ASCII character has a numeric code, e.g., the letter ‘h’ corresponds to the numeric value 104.) The difficulty, however, is that most ASCII values are not small enough—adding numbers in the range of 20 to 127 (ASCII values for letters, digits, and common symbols) to RGB values, which are in the range of 0 to 255, will produce a visible difference on the image. To solve this problem, we use the binary representation of the ASCII value since the 0s and 1s that make up a binary representation are small relative to the \texttt{uint8} range.
Binary representation

Our everyday number system is decimal, i.e., base 10. For example, the base 10 value 104 is “constructed” like this: $1 \cdot 10^2 + 0 \cdot 10^1 + 4 \cdot 10^0 = 104$. The binary, base 2, representation of the base 10 value 104 is $1101000$:

$$1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 104$$

The built-in function `dec2bin` does this conversion for us, returning a vector of type `char` (commonly called a `string`). For example, `dec2bin(104)` returns the `char` vector `'1101000'`. We will use strings of length 8 to store the binary representation, so the string for the value 104 should be `'01101000'`—the leading digit `0` is left-concatenated to `'1101000'` to get a length 8 string.\(^2\)

Splitting the binary representation to produce small values

We need small values to add to (or subtract from) the RGB values of an image, so we split the length 8 binary representation of each character into four parts, each of length 2. Consider the letter ‘h’:

<table>
<thead>
<tr>
<th>'h'</th>
<th>← The character</th>
</tr>
</thead>
<tbody>
<tr>
<td>104</td>
<td>← The ASCII value</td>
</tr>
<tr>
<td>'01101000'</td>
<td>← The binary representation</td>
</tr>
<tr>
<td>['01', '10', '10', '00']</td>
<td>← The split binary representation (4 parts)</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

In this way, any character in the ASCII set can be represented using four small base 10 values; they are

0, corresponding to `'00'`
1, corresponding to `'01'`
2, corresponding to `'10'`
3, corresponding to `'11'`

The built-in function `bin2dec` converts a binary string to a decimal value. For example, `bin2dec('11')` returns the numeric value 3 (type `double`).

How does a character “fit” in the pixels?

A character is now represented by four values; we will call them “quarter values.” Each pixel in a color image has three values: red, green, and blue. Therefore hiding one character in an image requires the modification of two pixels; we will use the red and green values of two adjacent pixels. Specifically, suppose a character is to be hidden starting at pixel \((i,j)\) of array \(X\), the array storing image data of type `uint8`, then

- the first quarter value is added to (or subtracted from) \(X(i,j,1)\),
- the second quarter value is added to (or subtracted from) \(X(i+1,j,1)\),
- the third quarter value is added to (or subtracted from) \(X(i,j,2)\),
- the fourth quarter value is added to (or subtracted from) \(X(i+1,j,2)\).

The choice between adding and subtracting depends on the value in the array. For example, if \(X(i,j,1)\) has the value 255, you can only subtract a quarter value from it, not add to it. Later on, in decrypting, you only need to determine the difference—absolute value—between the original and modified pixel values, so either adding or subtracting is fine for encryption provided that the range of the type `uint8` is handled correctly.

Hiding the entire message in the image

We assume that the user always (wisely) chooses an image large enough to accommodate the message to be encrypted. Given a message to be encrypted, we prepend and append it with the slash character ‘/’. For example, if the message “Top secret!” is to be encrypted, then we actually encrypt the string ‘/Top secret!/’. This means that the message to be encrypted cannot include the slash character, but

\(^2\)The ASCII set has only 128 characters so length 8 is more than enough to store the binary representation.
this makes decrypting easier later. The starting character, '/', is to be encrypted starting at a pixel randomly chosen from the top-left quarter of the image—randomly generate integer values in \([1, \text{ceil}(nr/2)]\) and \([1, \text{ceil}(nc/2)]\) where \(nr\) and \(nc\) are the number of rows and number of columns, respectively, of the image data array. Use the pixels to the right of the starting pixel, up to the second-last column, to continue encrypting the message character by character. If more rows of pixels are needed, skip two rows and then continue encrypting at the column index previously generated.

The figure on the right shows the locations of the modified pixels on one layer (e.g., red layer) of the 3-d array for encrypting a message 11 characters in length. Including the starting and ending slash characters, pixels for 13 characters are modified, shown in magenta. The randomly generated starting row index is 3 and the randomly generated starting column index is 5 for this example.

What to do

Implement the functions convertStr2bin, encrypt, and decrypt as specified using the method described above. The provided script encryptionExample demonstrates how to call encrypt and decrypt to hide and reveal a message, respectively. Note that in the script, the modified image containing the encryption is saved using the “bitmap” (bmp) format instead of the “jpeg” (jpg) format. The reason is that the jpeg format compresses data in a way that causes some loss of details. This does not work for us here because steganography is entirely based on small differences—we can’t afford to lose any details! Instead, the modified data is saved in the bitmap format, which incurs no loss.

The image files pond.jpg and pondEnc.bmp are available on the course website; try your decrypt function on these images to determine the secret message! Do not use cell array (to be introduced the week of 4/5) in this project.

function binMatrix = convertStr2bin(str)
% Create a character matrix to store the binary representation of a string.
% str: A string (1-d array of characters) of length n
% binMatrix: An (n+2) by 8 matrix of characters. Each row stores the binary
% representation of a character:
% Row 1 and row (n+2) each stores the binary representation of the character "/".
% Row i stores the binary representation of the str(i-1), i=2,3,...,n+1.

dec2bin(d) returns a string that is just long enough to store the binary representation of the numeric value in d, so you must prepend the returned strings with the digit ‘0’ as necessary to build strings of length 8. Can you identify any subtask that would make sense as a subfunction?

function xEnc = encrypt(x, str)
% Encrypt a string in an image.
% x: a uint8 array storing the data of a color image
% str: the string (1-d array of characters) to be encrypted. The slash
% character '/ ' is not allowed in str.
% xEnc: the uint8 array x modified to encrypt the string constructed
% by prepending and appending str with the slash character '/ '/.
% % Make effective use of convertStr2bin.

Carefully consider how to implement encrypt—identify subtasks and implement them as subfunctions. Taking the time to plan thoroughly will end up saving you time when you write and debug your code. The same is true for function decrypt specified below. You may find the built-in function strcmp helpful for comparing two strings, e.g., strcmp('01', '01') returns true because the two strings arguments are identical while strcmp('01', '1') returns false because the two string arguments are different (one is a length 2 string while the other is a length 1 string).
function strDecoded = decrypt(x, xEnc)
% Recover the string hidden in an image by comparing the original and modified images.
% x: uint8 array storing the data of the original color image
% xEnc: uint8 array storing the data of the image modified to encrypt a string
% strDecoded: the decrypted string

You will need to calculate the “absolute value” from uint8 subtractions, but note that a simple subtraction is not “safe” when working with uint8s—no negative value is possible and an underflow to zero can happen! Here is a neat way to calculate the difference (absolute value) between two uint8 numbers a and b: diff = (a-b) + (b-a). Think about why this works!\(^3\)

Submit your files convertStr2bin.m, encrypt.m, and decrypt.m on CMS.

\(^3\)An alternative way to get the difference between two uint8 values is to cast the type to double before you perform the subtraction and then use built-in function abs. But this approach has a high cost in terms of memory: a double uses eight times the memory of a uint8.