

CS1112 Fall 2022 Project 4 (Optional problem. This will not be graded.)

If you would like extra practice with images, complete the problem below. You will not submit your work on this problem as it is an optional problem, but it will be good practice for prelim 2.

4 Encryption and Decryption

People can send coded messages such that only the intended recipients who possess the "key" to unlock the message can understand it. A message can be words, numbers, an image, sounds, ..., etc., and a key can be as simple as a number, more complex such as an image, or a combination of different keys. For example, the Cesar Cipher is an encryption scheme where each letter in an alphabet is encoded as a letter further down the alphabet. The key is then simply the integer value of the shift. With the key 3, the message "I like fish" would be encrypted as "L olnh ilvk". Someone receiving the message "L olnh ilvk" can understand it easily if they knew that the key was 3, i.e., the message can be decrypted using the key 3 to become "I like fish".

In this problem, you will encrypt and decrypt a digital image using another image as the key. You will use the RGB values of the key and add them to the RGB values of the original image in order to do encryption. Analogously, you will subtract the RGB values of the key from an encrypted image in order to decrypt it. The arithmetic operations are carried out on the image pixel by pixel. This problem requires that you deal with the limits of the type `uint8`.

Download `p4files.zip` and extract the files into your current working directory in MATLAB. Included in the zip are two `png`¹ image files, one that is a coded image (`secretPhoto.png`) and the other is the key image (`key.png`) that was used to make the coded image. Later on, feel free to use your own images for testing! There are other files in the folder, which will be used for another problem later.

4.1 Decrypt

Implement function `decrypt` as specified:

```
function de = decrypt(A, key)
% Decrypt an encrypted image using a key image
% A: RGB data of an image, a 3-d uint8 array. The image was encrypted
%     using the key image.
% key: RGB data of the key image used to encrypt A. key is a 3-d uint8 array.
%
% de: RGB data of A decrypted using the key image. de is a 3-d uint8 array
%     the same size as A.
%
% To decrypt A, subtract the RGB values of key from the RGB values of A
% pixel by pixel. A and key may not have the same number of rows and
% columns. Suppose A has m rows and n columns and is smaller than or equal
% in size to key, then subtract the RGB values of the first m row and first
% n columns of key from the RGB values at the corresponding pixels of A.
% If A is bigger than key, then tile the key to decrypt A. Any underflow
% due to uint8 subtraction should wrap around, e.g., uint8(0)-uint8(3)
% should result in the decrypted value of uint8(253).
```

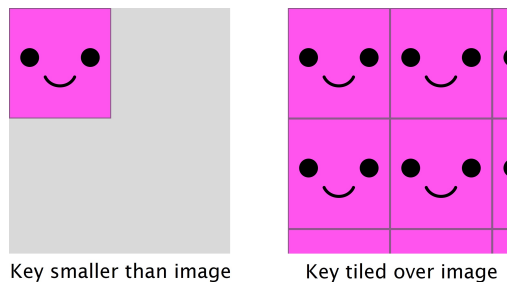
Data. The parameters are the `uint8` arrays storing the RGB data the image and the key, not the `png` image file names. To get the data into the Workspace for later calling your function, use `imread` in the Command Window, e.g.,

```
A= imread('secretPhoto.png'); % data of encrypted image
key= imread('key.png'); % data of key image
en= decrypt(A, key); % Call your function after you have completed it
```

Tiling. The term "corresponding pixels" means that you line up the key and the image at the top left corner; then pixel (i,j) of the image corresponds to pixel (i,j) of the key. If the key is smaller than the image, then you need to make a key big enough so that every image pixel has a corresponding pixel in the key. I.e.,

¹The code that you will write for encryption/decryption using `png` images is no different from that you would write had the image been in the `jpg` format. The reason we use the `png` format here is to avoid data loss due to the compression to the `jpg` format *after* encryption/decryption has been done.

you need to tile the given key so that there are at least as many rows and columns of pixels as the image. The diagram below illustrates how a key (smiley face on pink) is tiled so that an image that is bigger (gray area) can be covered. You can achieve tiling by actually creating a bigger array as the key. Alternatively, you can use function `rem` to cycle through the row (column) indices of the key as you work through the row (column) indices of the bigger image.



The encrypted image that we provided is smaller than the key, so you can get started with decryption easily. Later, after you have written the next function, `encrypt`, be sure to get an image larger than the key so that you can test that your tiling works.

Wraparound. Wraparound is one of several ways to deal with computation that results in values that would be “out of bounds” for a particular type (set). You have seen that MATLAB uses a different strategy, *saturation*, for the type `uint8`, i.e., it caps any overflow to 255 and any underflow to 0. However, saturation is *incompatible* with our decryption (encryption) scheme since the results from subtraction (addition) would not be unique, e.g., `uint8(5)-uint8(6)` and `uint8(5)-uint8(9)` in MATLAB both result in `uint8(0)`. That means that if your encryption results in saturation, then in decryption you would not be able to recover the original value! Therefore, we will use a different strategy, *wraparound*, to deal with underflow and overflow. With wraparound, treat all the values in a set as a cycle instead of a linear number line. Take the analog time clock as an example, the set of values for hours is `[1 2 ...12]` and any addition or subtraction that you do in regard to hours is done on a cycle, i.e., after 12h is 1h (not 13h). That’s *wraparound*—13h wraps around to become 1h. In the other direction, 0h wraps around to become 12h.

You need to write code for decryption (and later encryption) that achieves wraparound for the type `uint8`. I.e., after `uint8(255)` comes `uint8(0)`, then, `uint8(1)`, and so forth. For example, `uint8(254)+uint8(3)` should give `uint8(1)`. In the other direction, before `uint8(0)` is `uint8(255)`, and before that is `uint8(254)`, and so forth. For example, `uint8(1)-uint8(4)` should result in `uint8(253)`.

4.2 Encrypt

Implement function `encrypt` as specified:

```
function en = encrypt(A, key)
% Encrypt an image using a key image
% A:   RGB data of an image, a 3-d uint8 array.
% key: RGB data of the key image for encrypting A. key is a 3-d uint8 array.
%
% en:  RGB data of image A encrypted by key. en is a 3-d uint8 array the
%       same size as A
%
% To encrypt A, add the RGB values of key to the RGB values of A pixel by
% pixel. A and key may not have the same number of rows and columns.
% Suppose A has m rows and n columns and is smaller than or equal in size
% to key, then add the RGB values of the first m row and first n columns of
% key to the RGB values at the corresponding pixels of A. If A is bigger
% than key, then tile the key to encrypt A. Any overflow due to uint8
% addition should wrap around, e.g., uint8(255)+uint8(3) should result in
% the encrypted value of uint8(2).
```

Have fun encrypting and decrypting your own images!