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 and seek help from the course staff, 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 project will solidify your understanding of 2-dimensional and 3-dimensional arrays. In Part A, you will write a function to determine some properties of a matrix. Additionally, you will use MATLAB as a tool to solve a system of linear equations. In Part B, you will work with a video one frame at a time—each frame is a uint8 array. Pay attention to the difference between uint8 and MATLAB’s default type, double.

Part A of Project 4 (Problems 1 and 2) appears in a separate document.

3 Pixelation for Identity Protection

You have seen this computer graphics effect before: the faces of several people may be “pixelated,” or broken into blocks, in a documentary film in order to hide their identity. Take a look at the two demonstration video clips on the Projects page of the course website to see an example. The video file tas.avi is the original video while the file taspix.avi contains the same video but with the pixelation effect used to hide the faces of the people (four esteemed CS1112 consultants who do not need to hide!).

We use a MATLAB face detection function and you can see that it isn’t perfect. The function detects a face reliably when the face is vertical (not tilted), complete, and faces the camera squarely. You probably also saw that in a segment of the video the detector falsely detected two additional “faces” in the background. We are not concerned with the limitations of the given detector in this project; we will use the detector as is. We also provide for your use the functions for loading a video, getting an individual frame of a video, adding a frame into a video, and closing video processing.

You will implement two functions in this part of the project:

1. pixelate: this function pixelates a given array of image data.

2. pixelateFaces: partial code is given to deal with the initialization and closing of video processing. You will write code to deal with the individual frames and call your pixelate function to pixelate the detected faces.

Download the file p4bFiles.zip from the course website and extract all the files. You will add code to pixelate.m and pixelateFaces.m. Do not modify the given code in any of the files.

While you can develop the pixelate function on any version of MATLAB, on your own computer or a computer in the lab, the pixelateFaces function requires the use of the MATLAB Computer Vision System Toolbox Version 5 or later, which is available in the public labs but may not be available on the version of MATLAB on your own computer. To find out which version of MATLAB you have and which toolboxes are available, at the Command Window type ver. The following example interaction in the Command Window shows that on that particular machine the MATLAB Computer Vision System Toolbox Version 4.0 is available (see second last line of sample output below):
Unfortunately Version 4.0 is not enough; you need Version 5.0 or later, which is available in the campus labs that have MATLAB.

3.1 Pixelation

Implement the following function as specified:

```matlab
def pixelate(img, n)
    % Pixelate the image array img by breaking it into n-by-n blocks; all the
    % pixels in an individual block have the same color.
    % Parameters:
    % img: an image array (type uint8) to be pixelated
    % n: the number of blocks in the row and column dimensions.
    %     Assume that n is smaller than the number of rows and the number
    %     of columns in img. Assume that n is a reasonable integer value,
    %     say, n>=3 and n<=12
    % Return:
    % pixelatedImage: the pixelated image array (type uint8). This array
    %     has the same size as img.
    % Pixelation is done by splitting the array into n-by-n blocks of pixels.
    % For a 2-d array, calculate the average value for each block and then
    % assign to all pixels in the corresponding block in pixelatedImage that
    % average value. For a 3-d array, calculate the average for each layer in
    % the block separately and assign the averages to the corresponding layers
    % in the corresponding block in pixelatedImage.
    % Let nr and nc be the number of rows and number of columns, respectively,
    % in img. If nr is divisible by n then each block has the same number of
    % rows of pixels; otherwise the first n-1 rows of blocks each has the same
    % number of rows of pixels while the nth row of blocks has the remaining
    % rows of pixels. The number of columns of pixels for the blocks is
    % similarly determined.

    When the last row (column) of blocks has a size that is different from that of the first n-1 rows (columns) of blocks, minimize the difference. For example, if a 19-by-20 array is to be split into 5-by-5 blocks, the number of rows of pixels in the five rows of blocks should be 4, 4, 4, 4, 3 and not something like 3, 3, 3, 3, 7 or 2, 2, 2, 2, 11, etc.

    Here is an example: The picture on the left is a normal picture of a classy Collegetown establishment; the picture on the right is the pixelated version. The right picture was produced by calling the pixelate function with n=50.
```
Example code that you can use for a test run of your function:

```matlab
p = imread('duck.jpg'); % Pick your own picture
q = p(1:50, 1:34, :); % This example grabs 50-by-34 pixels. Try different sizes.
z = pixelate(q, 4);
imshow(z) % Does it look right? Next, actually look at the values
% in z by double-clicking on z in the Workspace pane or
% displaying part of z, e.g., disp(z(1:20, 1:12, 1))
```

### 3.2 Pixelate Faces

Now you will implement the main function that will work frame by frame to pixelate the faces that are found by the given face detector. Read the partially implemented function `pixelateFaces`; this is the function that you need to complete. Read the function header and comments of the other given files; you will call those functions so reading the comments is important (but you’re not responsible for the code in those files).

**Note:** The given code takes care of the initialization and closing of video processing. You will need to use some of the variables assigned in the given code: `numFrames`, `vidIn`, and `vidOut`. Your code needs to loop through the frame numbers from 1 to `numFrames` and at each iteration perform the following tasks:

- Get a frame from the video by calling the given function `getFrame`. The statement `img = getFrame(vidIn, k)` gets the kth frame from `vidIn` (the given variable that references the input video) and stores in variable `img` the array of image data (type `uint8`) for that frame.

- Look for faces in the frame by calling the detector: `bbox = findFaces(img)`
    The variable `bbox` then stores the data of the “bounding boxes” that correspond to the found faces. A bounding box is a rectangular area that bounds a face. Be sure to read the function header and comments in `findFaces.m` to learn how to use the values in the returned matrix.

- For each bounding box, i.e., for each face found, you have to determine the appropriate subarray that should be sent to the function `pixelate` for pixelation.

**Size of subarray** Let the height and width of the bounding box (number of rows of pixels and number of columns of pixels) be $h$ and $w$ and let the size of the subarray to be passed to `pixelate` be $nr$-by-$nc$. You need to find $nr$ such that it is the largest multiple of `numDivs` that is less than or
equal to $h$. Similarly, set $nc$ to be the largest multiple of $\text{numDivs}$ that is less than or equal to $w$. I.e., you need to find the largest subarray fitting inside the bounding box that would result in a pixelation with all blocks having the same size. For example, if the bounding box has height 125 and width 100 and $\text{numDivs}$ is 6, then $nr = 120$ and $nc = 96$.

**Location of subarray** Which part of the $nr$-by-$nc$ subarray within the bounding box should be sent to **pixelate**? The “middle part.” Continuing with the example above, let us suppose that the 125-by-100 bounding box has the upper left corner at row 5 column 150. Then the 120-by-96 subarray that should be sent to **pixelate** has its top left corner at row 7 (or row 8) and column 152.

**What if bounding boxes overlap?** Process each box individually, so that the pixelation result from the bounding box processed last overwrites the result from the previous bounding box processed in the overlapped area. Make sure that the pixelation for each bounding box is computed from the original image data.

- Add the frame that now has its faces pixelated to the output video by calling a given function: $\text{addFrame}(\text{vidOut}, \text{imgPix})$ where $\text{vidOut}$ is the given variable that references the output video and $\text{imgPix}$ is the image array of the frame but with pixelation done on the subarray(s) that correspond(s) to the detected face(s).
- Print a message to the Command Window to indicate to the user the progress, e.g., “Processing frame $x$ of $y$” or print the percentage of completion so far.

The provided code takes care of closing off video processing. After this function executes, you will see a new video file created as named by $\text{outputVideoName}$. You can play that video using any video player that accepts the .avi format.

For code development, be sure to use a short video no longer than a couple of seconds. The example video on the course website is about four seconds long and takes about a minute to process on a typical machine. You can run the **pixelateFaces** function like this:

```plaintext
pixelateFaces(‘tas.avi’, ’tasEdited.avi’, 6)
```

The above statement assumes that video file $\text{tas.avi}$ is in the current directory, calls the function to pixelate each face found using 6-by-6 blocks, and saves the resultant video in the file $\text{tasEdited.avi}$ in the current directory.

**Observations**

1. The sound is missing in my output video.
   Reason: In order to support multiple operating systems, we chose to forego the audio. Solution: If you have video editing software, you can retrieve the sound of the original video and plug it into the new video!

2. I want to edit my own video.
   Solution: Sure. We chose the .avi format which is supported in Windows, Mac, and Linux. You should make sure that the video you want to edit is .avi. If it’s not, you can use the following website for an easy and free conversion: [http://video.online-convert.com/convert-to-avi](http://video.online-convert.com/convert-to-avi). Of course, another video format may be supported by your specific operating system, but .avi is the only one that was tested for this project. Use at your own risk.

3. My video is taking forever to process
   Reason: video processing and face detection are not easy and are computationally intensive. Be sure to use very short video clips for program development. After you have the program working correctly, you can process a longer video, but plan to go outside and play while the video is processing . . .

4. This computer graphics stuff is awesome! Where can I learn more?
   Solution: Cornell offers several courses in computer graphics and computer vision, including CS 4620 Introduction to Computer Graphics and CS 4670 Introduction to Computer Vision. But first you will need to learn about data structures and object-oriented programming (CS 2110) and linear algebra if you wish to pursue this particular CS path.
Have fun developing this program and editing videos! Then submit your files `pixelate.m` and `pixelateFaces.m` on CMS.

*Final note:* You may have used vectorized code and the built-in functions `sum`. That’s fine for this part of the project. But some time before the next prelim, be sure to try solving this problem *without* vectorized code and without `sum` and `mean`. That would require more use of nested loops and allow you to practice basic array algorithms (such as traversing, summing rows and columns, etc.). It’s excellent practice for working with arrays and loops!