HW 1- Efficiency, Modular Programming, and Bad Puns

ENGR 3H

Due 5:00 PM, Wednesday, April 10

1 Let's Get Functional

1.1 Warm-Up (10 pts)

Write a function called `lyapunov.m` that takes as input two matrices and/or vectors, A and B, and returns the value C, where C is defined as:

$$-C = A^T B + BA \tag{1}$$

. The function should check if C can be computed, and if not, should return the message:

```
Matrix dimensions do not agree.
```

Hold on to this function; we'll use it again later in the course.

1.2 Image Blur and the Notorious C.F.G. (30 pts)

In image processing, a *convolutional filter* can be used to modify or extract features from an image. A filter applies an operation to a patch of an image, then moves on to the next patch, applying the same operation. Typically, this will consist of the summation of the elementwise multiplication of the filter with a given patch. As an example, consider a simple 2x2 filter F:

$$F = \begin{bmatrix} .25 & .4 \\ .1 & .25 \end{bmatrix}$$

. We'll apply this filter to the 3x3 matrix A:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

. To apply the filter, we start in the top left corner and "slide" it along, first left to right and then shifting down, to get the new values. For example, the first

filtered value would be equal to $.25(1) + .4(2) + .1(4) + .25(5) = 2.7$. We would then shift the filter over one spot and compute the next value: $.25(2) + .4(3) + .1(5) + .25(6) = 3.7$. The finished convolution would yield the new matrix:

$$\begin{bmatrix} 2.7 & 3.7 \\ 5.7 & 6.7 \end{bmatrix}$$

.

1.2.1

We're going to write some simple code to implement a type of filter known as a Gaussian filter to an image. You've actually probably used this kind of filter before if you've ever blurred an image. First, let's write a function called `create_gaussian` that takes as input an odd integer n and a standard deviation σ and returns the nxn Gaussian filter G. Values of the Gaussian filter decrease as the distance from the center is increased, according to the equation:

$$G_{i,j} = \frac{1}{2\pi\sigma} \exp - \frac{(i - y_{mid})^2 + (j - x_{mid})^2}{2\sigma^2}, \tag{2}$$

where y_{mid} and x_{mid} are the coordinates of the matrix's centerpoint (in MATLAB for a 3x3 matrix, for example, this would be $(2, 2)$). Note that the values of G should also be normalized such that $\sum G_{i,j} = 1$. For example, the 3x3 Gaussian filter with $\sigma = 1$ should be:

$$G = \begin{bmatrix} 0.0751 & 0.1238 & 0.0751 \\ 0.1238 & 0.2042 & 0.1238 \\ 0.0751 & 0.1238 & 0.0751 \end{bmatrix}$$

1.2.2

Now let's write a function called `apply_filter` that takes as input an nxn filter F and an lxw matrix M. The function should return the matrix M_f, the result of the filter being applied to M. This function should work for *any* nxn input filter F and any image/matrix M, provided that matrix has dimensions larger than or equal to n. For an added challenge, try making the code work for non-square filters as well (this is optional).

1.2.3

Time to put them together! Download the file "cfg.mat" from Gauchospace. Loading this file in MATLAB will provide you with the 128x128 matrix "cfg." Type "imshow(cfg)" to see the image in the file; now, write a script that creates a gaussian filter, applies it to this image, and displays the resulting version.

1.2.4

You may have noticed that the image you created in the previous step was smaller than the original image. Suppose instead we want to create an image that's the same size. To do this, we would first introduce "padding" to our matrix. This padding consists of rows and columns of zeros surrounding our original matrix so that the dimensions after applying the filter are the same as the original. Write a function called `zero_padding` that takes as input an image matrix A and an integer n, then appends n rows and columns of zeros and returns the resulting matrix. For example, if matrix A is

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix},$$

calling `zero_padding(A, 1)` should return:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 3 & 4 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

Use this function to modify the script you wrote in 1.2.3 to output a blurred image that is the same size as the original. Upload this script in addition to the functions.

2 Why We Write Efficient Code (10 pts)

In this problem, we're going to make use of a particularly useful analysis tool in MATLAB–the "Run and Time" button. Run and Time gives you a breakdown of where your code is spending a lot of time, which can give valuable insight into how to make it more efficient. We'll start with a toy problem to illustrate this point.

Write a function called `bad_exp.m` that takes a matrix, A, and an integer, n, and returns the value of A^n by multiplying A by itself $n-1$ times in a for loop. Now write a program that takes a matrix A and calculates A^n both by calling `bad_exp.m` and by computing it directly (to avoid having numerical issues, I recommend setting all of the entries of A to have magnitude less than 1). Use Run and Time for different values of n, starting at 2 and going up to 1,000,000. How does the time spent change (that is, how does the time spent on different parts of the code scale with N)? What does this suggest to you?