EE 046200 - Technion - Image Processing and Analysis

Computer Homework 2

Due Date: 02.06.22

Submission guidelines

READ THIS CAREFULLY

- **Please notice:** Some of the exercises contain questions on topics that are yet to be taught in the lecture or the frontal exercises. You may consider them as background or preparation questions to the topic before learning about it in class, or you may wait until the topic is taught, and solve only the questions on the topics you already learned.
- **Avoid unethical behavior.** This includes plagiarism, not giving credit to source code you decide to use, and false reporting of results. Consulting with friends is allowed and even recommended, but you must write the code on your own, independently of others. The staff will treat unethical behavior with the utmost severity. **!אנא המנעו מהתנהגות שאינה אתית והעתקות**
- Code submission in **Python only**. You can choose your working environment:
  - You can work in a Jupyter Notebook , locally with Anaconda (the course's computer HW will not require a GPU).
  - You can work in a Python IDE such as PyCharm or Visual Studio Code. Both also allow opening/editing Jupyter Notebooks.
- The exercise must be submitted **IN PAIRS** (unless the computer homework grader approved differently) until **Thursday 02.06.2022 at 23:55**.
- The exercise will be submitted via Moodle in the following form: You should submit two **separated** files:
  - A report file (visualizations, discussing the results and answering the questions) in a `.pdf` format, with the name `hw2_id1_id2.pdf` where `id1`, `id2` are the ID numbers of the submitting students.
    - Be precise, we expect on point answers. But don't be afraid to explain you statements (actually, we expect you to).
    - Even if the instructions says "Show/Display...", you still need to explain what are you showing and what can be seen.
    - No other file-types (`.docx` , `.html` , ...) will be accepted
  - A compressed `.zip` file, with the name: `hw2_id1_id2.zip` which contains:
    - A folder named `code` with all the code files inside (`.py` or `.ipynb` ONLY!)
      - The code should be reasonably documented, especially in places where non-trivial actions are performed.
      - Make sure to give a suitable title (informative and accurate) to each image or graph, and also to the axes. Ensure that graphs and images are displayed in a sufficient size to understand their content (and maintain the relationship between the axes - do not distort them).
    - A folder named `my_data`, with all the files required for the code to run (your own images/videos). make sure to refer to your input files in the code locally. i.e. (if the code is in 'code' directory, and the input file is in a parallel 'my_data' directory: `img = cv2.imread('../my_data/my_img.jpg')`
    - **DO NOT** include the given input data in the zip. The code should refer to the given input data as it is located in a folder named `given_data`. i.e.: `img = cv2.imread('../given_data/given_img.jpg')`
- If you submit your solution after the deadline, 4 points will be reduced automatically for each of the days that have passed since the submission date (unless you have approved it with the course staff before the submission date). Late submission will be done directly to the computer homework grader via mail, and not via Moodle.
- Several Python, numpy, openCV reference files are attached in the Moodle website, and you can of course also use the Internet's help.
- Questions about the **computer** exercise can be directed to the computer homework grader through the relevant Moodle forum or by email **and not during the workshop hours**.

General Notes:

The 'imshow' function:

Full name: `matplotlib.axes.Axes.imshow`

The 'imshow' function is used to display images. The function expects to get a matrix whose members are in "discrete" `unit8` format (in the range [0,255]) or in "continuous" `float` format (in the range [0,1]). the dynamic range is determined by the format. These formats are acceptable for images.

In [2]:
```python
# imports for the HW
import numpy as np
import matplotlib.pyplot as plt
import cv2
```

Part 1 - Creating a Panorama Using Motion Estimation

In This part we will learn how to create a panorama of images, meaning adding and "stitching" of different images. To do this we will use the correlation method in order to implement template matching (according to the principles presented in class, in exercise 4).

During this part we will use the music video of the song Corsica by Petru Guelfucci. You are encouraged to open the file `Corsica.mp4`, to watch the video and enjoy the music. :)

1.a - Find High Correlation Location:

In order to create the panorama image with good stitching between two images, we have to find similar regions of interest between the images. In order to achieve this, we will use the Correlation Index (convolution without the kernel mirroring). The Correlation Index is defined for one-dimensional signals $f[n], g[n]$ as:

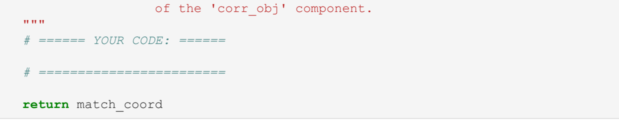
$$(f \star g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[m+n]$$

Meaning, the same as convolution, but without the mirroring.

Implement the `match_corr` function. The function accepts as input the 2d numpy array `corr_obj` containing an image of a certain component (e.g., an apple tree) and another 2d numpy array `img`, which must have equal or larger height and width in comparison to `corr_obj`. The `img` input is an image which contains in it the component from `corr_obj` (e.g. - an image of a graden with the same apple tree in it).

The function will perform 2d correlation between the input arrays and will return the location (indices) of `corr_obj`'s center using the coordinates of `img`.

For example, given the following `corr_obj` (left) and `img` (right):



The output of the function will be the 2d coordinates of the yellow dot in `img`.

In [9]:
```python
def match_corr(corr_obj, img):
    """
    return the center coordinates of the location of 'corr_obj' in 'img'.
    :param corr_obj: 2D numpy array of size [H_obj x W_obj]
                     containing an image of a component.
    :param img: 2D numpy array of size [H_img x W_img]
                where H_img >= H_obj and W_img>=W_obj,
                containing an image with the 'corr_obj' component in it.
    :return:
        match_coord: the two center coordinates in 'img'
                     of the 'corr_obj' component.
    """
    # ====== YOUR CODE: ======

    # ========================

    return match_coord
```

Notes:

- Use the `cv2.filter2D` method to perform the correlation.
- Note that the output image is the same size of the input image. Be sure to choose `borderType=cv2.BORDER_CONSTANT` as input in order to pad the source image with zeros (similar to 'same' convolution).
- In order to find the center of `corr_obj` inside `img` it is recommended to first find the maximal correlation value of the `corr_obj` image with itself - let's denote it as `max_obj`. After that, we'll find the point in `img` where the correlation value with `corr_obj` is the closest to `max_obj`.

1.b - Pre-Processing:

We will work on a section of 10 seconds, `04:10-04:20`, in which the camera moves horizontally. Load the `Corsica.mp4` frames in this time section (10s x 25FPS = 250frames). Transform the frames to grayscale and take only the lower two-thirds of each frame (excluding the singer from each frame). In addition, use only indices `7:627` for the width of each frame (excluding the black margins from each frame). From now on, whenever we adress the frames' height and width, we mean the height and width after the cut.

- You may use the `video_to_frames` function from your first python HW.

1.c - Creating the panorama base

1. Create the array in which the panorama will be inserted. The array will contain zeros. Its height will be the same as a frame's height, and its width will be 2.5 the width of a frame.
2. Select a reference frame, which will be the center of the panorama image, from the 10 seconds section mentioned. Make sure the frame you choose is not at the begining or at the end of the section. Display the original frame and the updated panorama image (zeros image with the reference frame at its center).
3. Select two additional frames - one earlier and one later than the reference frame (will now be called the early and late frame, respectively), both should still be included in the same 10 seconds section. Choose so there is some overlap between the selected frames and the reference frame. Display the frames you chose.

1.d - Frames matching

choose rectangular sub-images out of the early frame and the late frame. Choose the rectangles to have the same height as the original frames, and to contain the ovelapping area with the reference frame.

Apply the `match_corr` function twice:

- using the reference frame and the sub-image of the early frame
- using the reference frame and the sub-image of the late frame

Display the sub-images you chose (grayscale and in the same figure). Set the titles to be the respective output coordinates of the `match_corr` function.

1.e - It's panorama time!

Now that you have the appropriate coordinates for the two sub-images, you can evaluate the shift in the image between the early frame and the reference frame and between the late frame and the reference frame (mostly in the horizontal axis, but possibly also a little in the vertical axis - leading to some blank parts in the panorama image). Paste the early frame and the late frame (the full frames, and not just the sub-images), as part of the panorama image according to the displacement you found. Display the resulting panorama image. Are there problems in the final result? Explain.

Notes:

- Since we set the width of the panorama image to 2.5 the frame width, there may be parts of the early frame and the late frame that will not fit into the panorama array (the images edges), in this case the early / late frame width can be cut to fit the width of the panorama array. On the other hand - there can also be black margins at the sides of the result.
- Note that there will be overlapping pixels between the reference frame and the early frame and between the reference frame and the late frame. In these cases, set the pixel value in the panorama image as the average of the pixel values in the two frames.

Part 2 - Spatial Filtering and Morphological Operations

In This part we would like to examine a practical use of the morphological operations we saw in exercise 4 and the spatial filters we saw in exercise 2. We will extract from an image of a keyboard, an image that contains only the keys.

You are encouraged to read the following openCV tutorial about morphological operations https://docs.opencv.org/3.4/d9/d61/tutorial_py_morphological_ops.html

2.a - Morphological operations

1. Load and display the image `keyboard.jpg`.
2. Create two morphological kernels:
   A. A vertical line, 8 pixels long.
   B. A horizontal line, 8 pixels long.
3. Apply erosion on the image using each of the kernels **separately** and display the two results. What are the geometrical structures being conserved in each of the resulting images?
4. Sum the two images from the above section and display the summation image. Choose a threshold of `0.2*255` and transform the image to be binary - containing only `0` and `255` pixels. Display the result.

2.b - Median filtering

At this point we would like to use a median filter to isolate the keys. The keyboard keys are the connected components that contain the text. First perform a logical inversion (`NOT`) on the image from section C and then apply a median filter using a `9x9` kernel and the `cv2.medianBlur` function. Explain why a median filter is appropriate (i.e., why did we not choose a mean filter)?

2.c - Back to morphological operations

Now we want to place the keys better since it is possible that the connection image from section 2.b also contains the edges of the keys. Create a third square kernel, with a side of size 8 and apply erosion to the image from section 2.b using it. Present the result obtained.

2.d - Image sharpening and final thresholding

1. perform intersection between the result image from 2.c and the original image (e.g., by converting the binary image to `0,1` values and multiplying the images element-wise). Notice that multlipication is defined only for two arrays with the same type, so make sure both images are of `uint8` type.
2. Sharpen the image by filtering it using the filter kernel K and the openCV function `cv2.filter2D` where:

$$K = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

   Display the sharpened result. Why does this filter make the image sharper?
   - Note that `cv2.filter2D` performs correlation and not convolution, but in our case is doesn't matter because K is symmetric.
3. Finally, in order to get rid of the unnecessary background on each of the keys, choose a threshold value that gives a good result in your opinion and transform the image to be binary. Display the final binary result and specify the chosen threshold in the title. Did you succeed in creating an image that contains only the keys in your opinion? Are there problems in the final result? Explain.

Part 3 - Image Restoration

In this part we want to restore an image out of a noisy version of it. In class, you learned about image restoration as solutions to optimization problems. In this field we look at our images from a probabilistic point of view in order to restore blurred/noisy images.

During this part we will use the movie trailer of Flash Gordon. You are encouraged to open the file `Flash Gordon Trailer.mp4`, to watch the video and enjoy the great music by Queen. :)

In this question we will test two different restoration algorithms, but in order to do so, we must first create a noisy version of a given image.

3.a - Pre-processing - Creating a noisy image

- Load one frame from the video `Flash Gordon Trailer.mp4`. The frame must be taken from the time section `00:20-00:21`. You may use the `video_to_frames` function from your first python HW.
- Display this frame as a color image.
- Choose **one** of the color channels out of the chosen frame: the **red** channel or the **green** channel. Display the grayscale image of the chosen channel.
- Decrease the size of the image by a factor of 2 using `cv2.resize`. From now on we will use only this channel (after the resize) as our original image.
- Implement the `poisson_noisy_image` function, defined below, by following this procedure:
  Let a be the number of photons that have to arrive into the camera in order to be translated into one gray level. Now, we will create our Poisson noisy image (a.k.a shot noise) in a way which simulates realistic noise induced in an image taken by an optical camera (photon counting):
  1. transform the type of `X`, the input image, to `float` (the values of the image should still be in the range of `[0,255]`, just represented as `float` istead of `uint8`) and multiply the gray level values by a in order to transform the image to number of photons units.
  2. Create a new Poisson noisy image by applying `np.random.poisson` on your image. Using this command, the value of every pixel in the input (number of photons) is refered to as the mean of a Poisson-distributed random variable.
  3. Divide the resulting image by a in order to return the image to normal gray levels.
  4. Clip the image to `[0,255]` using `np.clip`, and transform the image type back to `uint8`.
  5. The noisy image you got will be the image Y.
- Make a noisy image out of the resized grayscale image using the `poisson_noisy_image` function with `a=3`.
- Display the noisy image result.

In [ ]:
```python
def poisson_noisy_image(X, a):
    """
    Creates a Poisson noisy image.
    :param X: The Original image. np array of size [H x W] and of type uint8.
    :param a: number of photons scalar factor
    :return:
        Y: The noisy image. np array of size [H x W] and of type uint8.
    """
    # ====== YOUR CODE: ======

    # ========================

    return Y
```

3.b - Denoise by L2

Let Y be a noisy image version of the image X. In order to restore X out of Y we would would to minimize the following expression (cost function):

$$\varepsilon^2\{\underline{X}\} = (\underline{X} - \underline{Y})^T(\underline{X} - \underline{Y}) + \lambda(D\underline{X})^T(D\underline{X})$$

Where $\underline{X}$ is a column-stack vector of the image X, $\underline{Y}$ is a column-stack vector of the noisy image Y, λ is the regularization parameter, and D is a sparse matix of the Laplacian operator, given by the kernel:

$$D_{kernel} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

In order to restore the source image we will apply an iterative process, based on the Steepest Descent algorithm. The update step of the algorithm is:

$$\underline{\hat{X}}_{k+1} = \underline{\hat{X}}_k - \mu_k \underline{G}_k = \underline{\hat{X}}_k - \mu_k((I + \lambda D^T D)\underline{\hat{X}}_k - \underline{Y})$$

Where I is the identity matrix and μ_k is the step size, which is determined by:

$$\mu_k = \frac{\underline{G}_k^T \underline{G}_k}{\underline{G}_k^T (I + \lambda D^T D)\underline{G}_k}$$

The process is initialized with $\underline{\hat{X}}_0 = \underline{Y}$.

Implement the algorithm described above in the following function, where `Err1,2` are defined as follows:

$$Err1\{\underline{\hat{X}}_k\} = (\underline{\hat{X}}_k - \underline{Y})^T(\underline{\hat{X}}_k - \underline{Y}) + \lambda(D\underline{\hat{X}}_k)^T(D\underline{\hat{X}}_k)$$

$$Err2\{\underline{\hat{X}}_k\} = (\underline{\hat{X}}_k - \underline{X})^T(\underline{\hat{X}}_k - \underline{X})$$

In [ ]:
```python
def denoise_by_l2(Y, X, num_iter, lambda_reg):
    """
    L2 image denoising.
    :param Y: The noisy image. np array of size [H x W]
    :param X: The Original image. np array of size [H x W]
    :param num_iter: the number of iterations for the algorithm perform
    :param lambda_reg: the regularization parameter
    :return:
        Xout: The restored image. np array of size [H x W]
        Err1: The error between Xk at every iteration and Y.
              np array of size [num_iter]
        Err2: The error between Xk at every iteration and X.
              np array of size [num_iter]
    """
    # ====== YOUR CODE: ======

    # ========================

    return Xout, Err1, Err2
```

Note that your algorithm uses the original image `X` only to calculate `Err2`, you sould not use it anywhere else in the algorithm!

Guidance: Note that your inputs `Y,X` and your output `Xout` are image matrices (2d numpy arrays), but $\underline{Y}, \underline{\hat{X}}, \underline{\hat{X}}_k$ are column-major order vectors. Use `npmatrix.flatten('F')` to create a column-order vector out of `npmatrix`. In addition, it is recommended to calculate the multlipication with D using convolution with the kernel, and **not** calculating the full Toeplitz matrix. Note that every time you encounter a multlipication with D in your calculation you must:

1. Transform the column vector into a matrix, using column-major order (`np.reshape(vector, newshape, order='F')`).
2. Convolve the matrix with the kernel using `cv2.filter2D` (note that the result has the same size as the input - 'same' convolution, and again - D_{kernel} is symmetric so it doesn't matter that `cv2.filter2D` performs correlation and not convolution).
3. Transform the resulting matrix back to a cloumn vector.

- Notice that for example the $\lambda D^T D\underline{\hat{X}}$ part requires you to do the above process twice.

Now, use the function you wrote on the noisy image you created in section 3.a. use `lambda_reg = 0.5` and `num_iters = 50`. Display the result of the restoration in your report.

In addition, display **on a single graph** a logarithmic plot of the errors `Err1` and `Err2` as a function of the iteration number, and explain.

3.c - Denoise by Total Variation

Now you will implement a restoration using a Total Variation prior. Meaning, you will work with the following cost function:

$$\varepsilon^2\{\underline{X}\} = (\underline{X} - \underline{Y})^T(\underline{X} - \underline{Y}) + \lambda \cdot TV\{\underline{X}\}$$

Where $TV\{X\}$ is the Total Variation function:

$$TV\{X\} = \sum_{x,y} |\nabla X| = \sum_{x,y} \sqrt{\left(\frac{\partial X}{\partial x}\right)^2 + \left(\frac{\partial X}{\partial y}\right)^2}$$

Where x, y are the 2D axes of the image. In order to perform restoration, we will again use an iterative process as in section 3.b, but now we will apply a gradient step appropriate to TV:

$$\hat{X}_{k+1} = \hat{X}_k + \frac{\mu_k}{2} U_k = \hat{X}_k + \frac{\mu_k}{2}\left(2\left(Y - \hat{X}_k\right) + \lambda \nabla \cdot \left(\frac{\nabla \hat{X}_k}{\sqrt{|\nabla \hat{X}_k|^2 + \epsilon_0^2}}\right)\right)$$

Note that there are no underlines in the above equation - here we use the images' matrices.

Implement this algorithm in the following function, where `Err1` is now defined as:

$$Err1\{\underline{\hat{X}}_k\} = (\underline{\hat{X}}_k - \underline{Y})^T(\underline{\hat{X}}_k - \underline{Y}) + \lambda \cdot TV\{\underline{\hat{X}}_k\}$$

In [ ]:
```python
def denoise_by_TV(Y, X, num_iter, lambda_reg, epsilon0):
    """
    TV image denoising.
    :param Y: The noisy image. np array of size [H x W]
    :param X: The Original image. np array of size [H x W]
    :param num_iter: the number of iterations for the algorithm perform
    :param lambda_reg: the regularization parameter
    :param: epsilon0: small scalar for numerical stability
    :return:
        Xout: The restored image. np array of size [H x W]
        Err1: The error between Xk at every iteration and Y.
              np array of size [num_iter]
        Err2: The error between Xk at every iteration and X.
              np array of size [num_iter]
    """
    # ====== YOUR CODE: ======

    # ========================

    return Xout, Err1, Err2
```

Guidance: Find the image's derivatives using `np.gradient`. think about how to use this function in order to also calculate a divergence.

Initilazation and hyperparameters:

- $\underline{\hat{X}}_0 = \underline{Y}$
- constant step size: $\mu_k = \mu = 150\epsilon_0$

Now, use the function you wrote on the noisy image you created in section 3.a. use `lambda_reg = 20` and `num_iters = 200`. Try some different values for ϵ_0 in the range $[10^{-7}, 10^{-3}]$, and choose a compatible one (by looking at the visual result and by making sure that that the error plot reaches convergence and a low value). Denote the chosen ϵ_0 in your report. After choosing ϵ_0, display the restored image in your report.

In addition, display **on a single graph** a logarithmic plot of the errors `Err1` and `Err2` as a function of the iteration number, and explain.

3.d - Results analysis

Express your opinion regarding the results. Present the L2 and TV restorations side by side and the errors plots side by side. Compare the results quantitivly (using `Err1,2` values) and qualitativly (by looking at the restored images). Do your results fit the theory taught in class?

3.e - From synthetic to natural

Up to this point we worked with a synthetic image (created by computer graphics). We now want to test the performance of both algorithms using more "natural" images.

Choose another frame from the video `Flash Gordon Trailer.mp4`. The frame must be taken from the time section `00:38-00:39` and must contain a natural image. Choose one of the color channels of the frame (Red or Green), decrease the size of the image by a factor of 2, and make a noisy image (according to the instructions in section 3.a). Repeat the denoising processes (1.b, 1.c). Display the restored images and the errors plots in your report. Explain - what are the differences examining the natural images results in comparison to the synthetic ones? Discuss `Err1,2` values and the restored images quality.

Credits

- Icons from Icon8.com - https://icons8.com