Fall 2022 CS543/ECE549

Assignment 2: Fourier-based Alignment and Finding Covariant Neighborhoods

Due date: Mon, October 9, 11:59:59 PM

Part 1: Fourier-based color channel alignment

For the first part of this assignment, we will revisit the color channel alignment that you performed in Assignment 1. The goal in this assignment is to perform color channel alignment using the Fourier transform. As I said in lecture (and in notes, and on slides), convolution in the spatial domain translates to multiplication in the frequency domain. Further, the Fast Fourier Transform algorithm computes a transform in `O(N M log N M)` operations for an `N` by `M` image. As a result, Fourier-based alignment may provide an efficient alternative to sliding window alignment approaches for high-resolution images.

Similarly to Assignment 1, you will perform color channel alignment on the same set of six low-resolution input images here and three high-resolution images here. You can use the same preprocessing from Assignment 1 to split the data into individual color channels. You should use only the original input scale (not the multiscale pyramid from Assignment 1) for both high-resolution and low-resolution images in Fourier-based alignment.

Algorithm outline

The Fourier-based alignment algorithm consists of the following steps:

1. For two color channels C1 and C2, compute corresponding Fourier transforms FT1 and FT2.
2. Compute the conjugate of FT2 (denoted as FT2*- if you don't remember your complex numbers, look this up!), and compute the product of FT1 and FT2*.
3. Take the inverse Fourier transform of this product and find the location of the maximum value in the output image. Use the displacement of the maximum value to obtain the offset of C2 from C1.

To colorize a full image, you will need to choose a base color channel, and run the above algorithm twice to align the other two channels to the base. For further details of the alignment algorithm, see section 9.1.2 of Computer Vision: Algorithms and Applications, 2nd ed.

Color channel preprocessing. Applying the Fourier-based alignment to the image color channels directly may not be sufficient to align all the images. To address any faulty alignments, try sharpening the inputs or applying a small Laplacian of Gaussian filter to highlight edges in each color channel.

Implementation Details

You should implement your algorithm using standard libraries in Python. To compute the 2D Fourier transforms you should use the `np.fft.fft2` function followed by the `np.fft.fftshift` function to shift components for better visualization. You can use `np.conjugate` to take the conjugate of a transform, and you should compute inverse transforms using the `np.fft.ifft2` function. Finally, you can use `scipy.ndimage.gaussian_filter` or `cv2.filter2D` for filter-based preprocessing of input channels.

In addition to the final aligned images, we will ask you to include visualization of the inverse Fourier transform outputs you used to find the offset for each channel. You can use `matplotlib.pyplot.imshow` to visualize the output. Make sure that the plots are clear and properly scaled so that you can see the maximum response region.

Part 1 Submission Checklist

In your report (based on this template), you should provide the following for each of the six low-resolution and three high-resolution images:

- Final aligned output image
- Displacements for color channels
- Inverse Fourier transform output visualization for both channel alignments ***without*** preprocessing
- Inverse Fourier transform output visualization for both channel alignments ***with*** any sharpening or filter-based preprocessing you applied to color channels

You should also include the following discussion:

- Describe any preprocessing you used on the color channels to improve alignment and how it changed the outputs
- Measure the Fourier-based alignment runtime for high-resolution images (you can use the python `time` module again). How does the runtime of the Fourier-based alignment compare to the basic and multiscale alignment you used in Assignment 1?

Part 2: Scale-space blob construction

The goal of Part 2 of the assignment is to build a blob coordinate system as discussed in lecture (and slides, and notes!).

Data

Demonstrate your code works on at least four images of your own choosing.

Algorithm outline

1. Find corners using a Harris corner detector. **You may use a library based corner detector** if you wish, though reading the manual will likely take as long as building your own.
2. For each corner:
    - At each corner location, apply a set of scale normalized Laplacian of Gaussian filters at several different scales.
    - Find the scale that gives the maximum absolute value response from the LOG filter, using non-linear interpolation between scale values.
    - Within the window defined by that scale, compute the most common orientation.
    - Mark on the image:
        1. The location of the corner, with an x
        2. The scale of the LOG at the corner, with a circle whose radius is the scale and whose center is the corner.
        3. The orientation of the window, with an arrow pointing from the corner in the direction of the orientation.

Demonstrating your method works

For each of your four chosen images you should show results on:

- The base image
- The image shifted about 20% to the left and cropped
- The image shifted about 20% to the right and cropped
- The image rotated by 90 degrees counterclockwise
- The image rotated by 90 degrees clockwise
- The image enlarged by a factor of 2 and center cropped

The expected behavior is that:

- When the image shifts, the circles shift, but the size and orientation does not change.
- When the image is rotated, the orientations rotate but the size does not change.
- When the image is scaled, the size changes but the orientations do not.

You **MUST** display this behavior for four images. The argument that your images display this behavior because they don't have any circles on them, and still don't after shift, rotate and scale, **will not work**.

Possible problem: if you don't scale normalize your LoG filter, your scales will be biased to be too small; this is usually pretty obvious, and we'll look for it.

Detailed instructions

- Convert images to grayscale. Then rescale the intensities to between 0 and 1 (simply divide them by 255 should do the trick).

- For creating the Laplacian filter, use the `scipy.ndimage.filters.gaussian_laplace` function. Pay careful attention to setting the right filter mask size.

- It is relatively inefficient to repeatedly filter the image with a kernel of increasing size. Instead of increasing the kernel size by a factor of k, you should downsample the image by a factor 1/k. In that case, you will have to upsample the result or do some interpolation in order to find maxima in scale space.
- **Hint 2:** Use `skimage.transform.resize` to help preserve the intensity values of the array.
- You have to choose the initial scale, the factor k by which the scale is multiplied each time, and the number of levels in the scale space. I typically set the initial scale to 2, and use 10 to 15 levels in the scale pyramid. The multiplication factor should depend on the largest scale at which you want regions to be detected.

- To display the detected regions as circles, you can use this function (or feel free to search for a suitable Python function or write your own). **Hint:** Don't forget that there is a multiplication factor that relates the scale at which a region is detected to the radius of the circle that most closely "approximates" the region.

Part 2 Submission Checklist

In your report (based on this template) you should provide the following for ***4 different examples*** 4 of your own:

- original image
- each of the five modified images (shift, rotate, scale) above

You should also include the following:

- Explanation of any "interesting" implementation choices that you made

Submission Instructions

As before, you must turn in both your report and your code. You should use the provided template.

To submit this assignment, you must upload the following files on **Canvas**:

1. Your code in two separate files for part 1 and part 2. The filenames should be **lastname_firstname_a2_p1.py** and **lastname_firstname_a2_p2.py**. We prefer that you upload .py python files, but if you use a Python notebook, make sure you upload both the original .ipynb file and an exported PDF of the notebook.
2. A brief report **in a single PDF file** with all your results and discussion following this **template**. The filename should be **lastname_firstname_a2.pdf**.
3. All your output images and visualizations **in a single zip file**. The filename should be **lastname_firstname_a2.zip**. Note that this zip file is for backup documentation only, in case we cannot see the images in your PDF report clearly enough. **You will not receive credit for any output images that are part of the zip file but are not shown (in some form) in the report PDF.**

Please refer to course policies on academic honesty, collaboration, late days, etc.

Further References

- Szeliski, Richard. Computer vision: algorithms and applications. Springer Nature, 2022. See pp. 563-566 for Fourier-based alignment.
- Scipy Gaussian Filter
- OpenCV Filter2D
- Sample Harris detector using scikit-image.
- Blob detection on Wikipedia.
- D. Lowe, "Distinctive image features from scale-invariant keypoints," International Journal of Computer Vision, 60 (2), pp. 91-110, 2004. This paper contains details about efficient implementation of a Difference-of-Gaussians scale space.
- T. Lindeberg, "Feature detection with automatic scale selection," International Journal of Computer Vision 30 (2), pp. 77-116, 1998. This is advanced reading for those of you who are *really* interested in the gory mathematical details.