Fall 2022 CS543/ECE549

Assignment 2: Fourier-based Alignment and Scale-Space Blob Detection

Due date: Thursday, October 6, 11:59:59 PM

- Part 1: Fourier-based Alignment
  - Alignment Algorithm
  - Implementation Details
  - Submission Checklist
- Part 2: Scale-space blob detection
  - Data and starter code
  - Blob Detection Algorithm
  - Extra Credit
  - Submission Checklist
- Submission Instructions
- Further References

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 discussed in this lecture, 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*), 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 detection

The goal of Part 2 of the assignment is to implement a Laplacian blob detector as discussed in the this lecture.

Data and starter code

This zip file contains starter code, four test images, and sample output images for reference. Keep in mind that your output may look different depending on your threshold, range of scales, and other implementation details. In addition to the images provided, also **run your code on at least four images of your own choosing**.

Algorithm outline

1. Generate a Laplacian of Gaussian filter.
2. Build a Laplacian scale space, starting with some initial scale and going for n iterations:
    1. Filter image with scale-normalized Laplacian at current scale.
    2. Save square of Laplacian response for current level of scale space.
    3. Increase scale by a factor k.
3. Perform nonmaximum suppression in scale space.
4. Display resulting circles at their characteristic scales.

Detailed instructions

- Don't forget to 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. **For full credit, you should turn in both implementations: one that increases filter size, and one that downsamples the image.** In your report, list the running times for both versions of the algorithm and discuss differences (if any) in the detector output. For timing, use `time.time()`.

  Hint 1: think about whether you still need scale normalization when you downsample the image instead of increasing the scale of the filter.

  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.

- You may want to use a three-dimensional array to represent your scale space. It would be declared as follows:

```
scale_space = numpy.empty((h,w,n)) # [h,w] - dimensions of image, n - number of levels in scale space
```

Then `scale_space[:,:,i]` would give you the `i`-th level of the scale space. Alternatively, if you are storing different levels of the scale pyramid at different resolutions, you may want to use an NumPy object array, where each "slot" can accommodate a different data type or a matrix of different dimensions. Here is how you would use it:

```
scale_space = numpy.empty(n, dtype=object) # creates an object array with n "slots"
scale_space[i] = my_matrix # store a matrix at level i
```

- To perform nonmaximum suppression in scale space, you should first do nonmaximum suppression in each 2D slice separately. For this, you may find functions `scipy.ndimage.filters.rank_filter` or `scipy.ndimage.filters.generic_filter` useful. Play around with these functions, and try to find the one that works the fastest. To extract the final nonzero values (corresponding to detected regions), you may want to use the `numpy.clip` function.

- You also have to set a threshold on the squared Laplacian response above which to report region detections. You should play around with different values and choose one you like best. To extract values above the threshold, you could use the `numpy.where` function.

- 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.

Extra Credit

- Implement the difference-of-Gaussian pyramid as mentioned in class and described in David Lowe's paper. Compare the results and the running time to the direct Laplacian implementation.

- Implement the affine adaptation step to turn circular blobs into ellipses as shown in the lecture (just one iteration is sufficient). The selection of the correct window function is essential here. You should use a Gaussian window that is a factor of 1.5 or 2 larger than the characteristic scale of the blob. Note that the lecture slides show how to find the relative shape of the second moment ellipse, but not the absolute scale (i.e., the axis lengths are defined up to some arbitrary constant multiplier). A good choice for the absolute scale is to set the sum of the major and minor axis half-lengths to the diameter of the corresponding Laplacian circle. To display the resulting ellipses, you should modify the circle-drawing function or look for a better function in the `matplotlib` documentation or on the Internet.

- The Laplacian has a strong response not only at blobs, but also along edges. However, recall from the class lecture that edge points are not "repeatable". So, implement an additional thresholding step that computes the Harris response at each detected Laplacian region and rejects the regions that have only one dominant gradient orientation (i.e., regions along edges). If you have implemented the affine adaptation step, these would be the regions whose characteristic ellipses are close to being degenerate (i.e., one of the eigenvalues is close to zero). Show both "before" and "after" detection results.

Part 2 Submission Checklist

In your report (based on this template) you should provide the following for ***8 different examples*** (4 provided, 4 of your own):

- original image
- output of your circle detector on the image
- running time for the "efficient" implementation on this image
- running time for the "inefficient" implementation on this image

You should also include the following:

- Explanation of any "interesting" implementation choices that you made
- Discussion of optimal parameter values or ones you have tried

Bonus

- Discussion and results of any extensions or bonus features you have implemented

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.