ECE549 / CS543 Computer Vision: Assignment 2

Instructions

1. Assignment is due at **11:59:59 PM on Thursday, March 12 2020**.
2. Course policies: http://saurabhg.web.illinois.edu/teaching/ece549/sp2020/policies.html.
3. Starter code, and data for this assignment is available at http://saurabhg.web.illinois.edu/teaching/ece549/sp2020/mp/mp2.zip.
4. Submission instructions:
   (a) A single `.pdf` report that contains your work for Q1, Q2 and Q3. For Q1(a) you can present your responses in the space provided on page 2 (directly typing on PDF, or via LaTeX, or by hand-writing on a print out and scanning it). For rest of the assignment, your response should be electronic (no handwritten responses allowed). You should respond to the questions 1(b), 1(c), 2(a), 2(b), 2(c), 2(d), 3(a), 3(b), 3(c) individually and include images, plots and metrics as necessary. Your response in the PDF report should be self-contained. It should include all the output you want us to look at. You will not receive credit for any results you have obtained, but failed to include directly in the PDF report file. PDF file will need to be submitted to https://www.gradescope.com (Entry Code: **MVZDNW**), and you will need to tag your PDF with where your response to each of the question is.
   (b) You also need to submit code for all questions in the form of a single **`.zip`** file. Code will need to be submitted to compass2g.
   (c) The LaTeX source is available: http://saurabhg.web.illinois.edu/teaching/ece549/sp2020/mp/mp2-latex.zip.
   (d) We reserve the right to take off points for not following submission instructions.

Change log

v0	02/24/2020	Creation.

1. **Dynamic Perspective [20 pts].** In this question, we will simulate optical flow induced on the image of a static scene due to camera motion. You can review these concepts from lecture 5.

   (a) **[10 pts]** Assuming a camera moving with translation velocity of $\mathbf{t}$ and angular velocity of ω. Derive the equation that governs the optical flow at a pixel (x, y) in terms of the focal length f, and the depth of the point $Z(x, y)$. Note that a point (X, Y, Z) in the world projects to $\left(f\frac{X}{Z}, f\frac{Y}{Z}\right)$ in the image.

(b) **[10 pts]** Next, we will try to visualize the optical flow induced on the image of a static scene due to camera motion. We will build off started code in `dynamic_perspective_starter.ipynb` (download from http://saurabhg.web.illinois.edu/teaching/ece549/sp2020/mp/mp2.zip). We have implemented two simple scenes, that of a vertical wall directly in front of the camera (`get_wall_z_image`), and that of a camera overlooking a horizontal plane (`get_road_z_image`). Your task is to use these scenes to visualize the induced optical flow for the following situations:

i. Looking forward on a horizontal plane while driving on a flat road.
ii. Sitting in a train and looking out over a flat field from a side window.
iii. Flying into a wall head-on.
iv. Flying into a wall but also translating horizontally, and vertically.
v. Counter-clockwise rotating in front of a wall about the Y-axis.

You should pick the appropriate scene, $\mathbf{t}$ and ω, and visualize the induced optical flow (you can use the `plot_optical_flow` function). Include the generated optical flow in your report. Note that the origin (for the perspective projection equations) is at the center of the camera, and this is different from the origin for images in `numpy`.

(c) **[10 pts] Extra Credit.** There aren't as many easy extra credit opportunities for this problem. However, we provide some suggestions below. In your report clearly identify what you did, and include the results you get.

- We generated some easy scenes for you for parts above. You could try to generate some other scenes by yourself. For example, you could simulate an aircraft heading down towards the ground at an angle of 45°, or some other interesting and instructive cases.
- For the special case of pure camera rotation, we don't need to know scene geometry to simulate optical flow. This can be used to generate 'virtual views' that depict how the scene would look like if we were to rotate the camera. There are more direct ways of doing this, that we will talk about later in the course. In the meanwhile, you could use optical flow information to generate such virtual views. Rather than computing a *forward flow*, it is more useful to compute the *backward flow* (*i.e.* the flow from the desired image to the original image). For each pixel in the desired image, this backward flow points to the location (in the original image) from which the pixel value should be 'copied' in order to populate the pixel value in the desired image. In general, this location may not fall on an exact pixel, and you will need to employ interpolation techniques to lookup values that fall in-between pixels. Further, note that the equations we derived make differential assumptions. So you will need to apply them iteratively multiple times for small δt times, to get an accurate rotated image.
- We also discussed in class how estimated optical flow can be used to compute time to collision when moving towards a fronto-parallel wall. You could try to capture some video footage and use it with off-the-shelf optical flow estimation algorithms to see how accurate such an estimate is.

2. **Contour Detection [30 pts].** In this problem we will build a basic contour detector. We will work with some images from the BSDS dataset [1], and benchmark the performance of our contour detector against human annotations. You can review the basic concepts from lecture 8.

We will generate a per-pixel boundary score. We will start from a bare bone edge detector that simply uses the gradient magnitude as the boundary score. We will add non-maximum suppression, image smoothing, and optionally additional bells and whistles. We have provided some starter code, images from the BSDS dataset and evaluation code. Note that we are using a faster approximate version of the evaluation code, so metrics here won't be directly comparable to ones reported in papers.

Preliminaries. Download the starter code, images and evaluation code from http://saurabhg.web.illinois.edu/teaching/ece549/sp2020/mp/mp2.zip (see `contour-data`, `contour_demo.py`). We have implemented a contour detector that uses the magnitude of the local image gradient as the boundary score. This gives us overall max F-score, average max F-score and AP of $0.51, 0.56, 0.41$ respectively. Reproduce these results by running `contour_demo.py`. Confirm your setup by matching these results. When you run `contour_demo.py`, it saves the output contours in the folder `output/demo`, prints out the 3 metrics, and produces a precision-recall plots at `contour-output/demo_pr.pdf`. Overall max F-score is the most important metric, but we will look at all three.

(a) **[5 pts] Warm-up**. As you visualize the produced edges, you will notice artifacts at image boundaries. Modify how the convolution is being done to minimize these artifacts.

(b) **[10 pts] Smoothing.** Next, notice that we are using $[-1, 0, 1]$ filters for computing the gradients, and they are susceptible to noise. Use derivative of Gaussian filters to obtain more robust estimates of the gradient. Experiment with different sigma for this Gaussian filtering and pick the one that works the best.

(c) **[15 pts] Non-maximum Suppression.** The current code does not produce thin edges. Implement non-maximum suppression, where we look at the gradient magnitude at the two neighbours in the direction perpendicular to the edge. We suppress the output at the current pixel if the output at the current pixel is not more than at the neighbors. You will have to compute the orientation of the contour (using the X and Y gradients), and implement interpolation to lookup values at the neighbouring pixels.

(d) **[Upto 10 pts] Extra Credit.** You should implement other modifications to get this contour detector to work even better. Here are some suggestions: compute edges at multiple different scales, use color information, propagate strength along a contiguous contour, *etc.* You are welcome to read and implement ideas from papers on this topic.

For each of the modifications above, your report should include:

- key implementation details focusing on the non-trivial aspects, hyper-parameters that worked the best,
- contour quality performance metrics before and after the modification (also include the PR plot),
- impact of modification on run-time,
- visual examples that show the effects of the modification on two to three images.

3. **Scale-space blob detection [30 pts][1].** The goal of this problem is to implement a Laplacian blob detector as discussed in lecture 10.

Algorithm Outline.

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

Test Images. We are providing eight images (four along with sample outputs for reference) that you should report results on. See folders `blobs-data` and `blobs-ref-output` at http://saurabhg.web.illinois.edu/teaching/ece549/sp2020/mp/mp2.zip. Keep in mind, though, that your output may look different from the reference output depending on your threshold, range of scales, and other implementation details.

(a) **[15 pts] Basic Implementation.** Implement the Laplacian blob detector as described above. In your report, include the output of your blob detector on all the eight provided images. Explain any 'interesting' implementation choices that you made. Describe the parameter values you tried and which ones were optimal. Here are some tips that may be useful.

- You can work with grayscale images.
- You may find the `scipy.ndimage.filters.gaussian_laplace` function useful.
- You will 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. Consider using an initial scale of 2, and 10 to 15 levels in the scale pyramid. The multiplication factor would depend on the largest scale at which you want regions to be detected.

[1] Adapted from Lana Lazebnik.

- 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.
- 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.
- Display resulting circles at their characteristic scales. To display the detected regions as circles, you can use code in `blobs_show_all_circles.py`. 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.

(b) **[15 pts] Efficient Implementation.** 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 your report, include the output of this more efficient implementation on all the images, and discuss differences in output from the basic implementation. Also report and compare the running times for both the basic and this efficient implementation. Discuss parameter choices, and any interesting implementation choices.

- You will have to upsample the result / do some interpolation before finding the maxima in scale space.
- Think about whether you still need scale normalization when you downsample the image instead of increasing the scale of the filter.
- You may find the `skimage.transform.resize` function useful.

(c) **[Upto 10 pts] Extra Credit.**

i. Implement the difference-of-Gaussian pyramid as mentioned in class and described in the SIFT paper [2]. Compare the results and the running time to the direct Laplacian implementation.

ii. 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). Show both 'before' and 'after' detection results.

References

[1] D. Martin, C. Fowlkes, D. Tal, and J. Malik. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *ICCV*, volume 2, pages 416–423, July 2001.

[2] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.