ECE4513 - Image Processing and Computer Vision

Assignment #3 Instructor: Zhen Li Due on 2022/12/25 23:59 pm

Part I Written Exercises

Problem 1 (10 points)

Show that forming unweighted local averages, which yields an operation of the form

$$\mathcal{R}_{ij} = \frac{1}{(2k+1)^2} \sum_{u=i-k}^{u=i+k} \sum_{v=j-k}^{v=j+k} \mathcal{F}_{uv}$$

is a convolution. What is the kernel of this convolution?

Problem 2 (10 points)

Each pixel value in 500×500 pixel image I is an independent, normally distributed random variable with zero mean and standard deviation one. I is convolved with the $(2k + 1) \times (2k + 1)$ kernel G. What is the covariance of pixel values in the result? There are two ways to do this; on a case-by-case basis (*e.g.*, at points that are greater than $2k + 1$ apart in either the x or y direction, the values are clearly independent) or in one fell swoop. Don't worry about the pixel values at the boundary.

Submission List:

- All answers in the file *your_school_id_name.pdf* (*e.g., 1190190xx_San_Zhang.pdf*).

Part II Programming Exercises

Problem 1 Image Pyramid (20 points)

You can only use the following Python libraries: OpenCV, numpy, math, and matplotlib.

Submission List:

- Codes for the implementation and result display in part **(a)** and **(b)** in the file *main.py*.
- All code files in the folder *prob_1*.
- Display the processing results and write your explanations for part **(a)** and **(b)** in the *your_school_id_name.pdf*. (*e.g., 1190190xx_San_Zhang.pdf*).

Choose an image that has an interesting variety of texture (from web or your own images). The image should be at least 640 × 480 pixels and converted to grayscale. Write code for a Gaussian and Laplacian pyramid of level *N*. In each level, the resolution should be reduced by a factor of 2. Show the pyramids for your chosen image and include the code in your write-up.

(a) Display a Gaussian and Laplacian pyramid of level 5 (using your code). It should be formatted similar to the figure below.

(b) Display the FFT amplitudes of your Gaussian/Laplacian pyramids. Appropriate display ranges should be chosen so that the changes in frequency in different levels of the pyramid are clearly visible. Explain what the Laplacian and Gaussian pyramids are doing in terms of frequency.

Your displayed images for the Gaussian and Laplacian pyramids should look something like this:

Problem 2 Edge Detection (20 points)

You can only use the following Python libraries: OpenCV, numpy, math, and matplotlib.

Submission List:

- Codes for the implementation and displaying results in part **(a)** and **(b)** in the folder *solutions*.
- All code files in the folder *prob_2*.
- Display the results and write explanations for part **(a)** and **(b)** in the *your_school_id_name.pdf*. (*e.g., 1190190xx_San_Zhang.pdf*).

The main steps of edge detection are:

(1) assign a score to each pixel;

(2) find local maxima along the direction perpendicular to the edge.

Sometimes a third step is performed where local evidence is propagated so that long contours are more confident or strong edges boost the confidence of nearby weak edges. Optionally, a thresholding step can then convert from soft boundaries to hard binary boundaries.

We have provided the test images with ground truth from BSDS500 in the folder *data* along with some code for evaluation in the folder *evaluateBSDS500*. Your job is to build a simple gradient-based edge detector, extend it using multiple oriented filters, and then describe other possible improvements. Details of your write-up are towards the end.

(a) Build a simple gradient-based edge detector that includes the following functions:

```
def gradientMagnitude(im, sigma):
    ...
    return mag, theta
```

This function should take an RGB image as input, smooth the image with Gaussian std=sigma, compute the *x* and *y* gradient values of the smoothed image, and output image maps of the gradient magnitude and orientation at each pixel. You can compute the gradient magnitude of an RGB image by taking the *L*2-norm of the R, G, and B gradients. The orientation can be computed from the channel corresponding to the largest gradient magnitude. The overall gradient magnitude is the *L*2-norm of the *x* and *y* gradients. mag and theta should be the same size as im.

```
def edgeGradient(im):
    ...
    return bmap
```

This function should use `gradientMagnitude` to compute a soft boundary map and then perform non-maxima suppression. For this assignment, it is acceptable to perform non-maxima suppression by retaining only the magnitudes along the binary edges produce by the Canny edge detector in OpenCV: `cv2.Canny(im, thresh1, thresh2)`. You may obtain better results by writing a non-maxima suppression algorithm that uses your own estimates of the magnitude and orientation. If desired, the boundary scores can be rescaled, *e.g.*, by raising to an exponent: `mag2 = mag**0.7`, which is primarily useful for visualization.

Evaluate using the codes in the folder *evaluateBSDS500* and record the overall and average F-scores. Please setup the evaluation code following the instruction file *README.md* in the folder *evaluateBSDS500*. You may run the evaluation attempt using the folder *bench* to check if the setup is successful. Please put your final predictions in the folder *data/png/test*.

(b) Try to improve your results using a set of oriented filters, rather than the simple derivative of Gaussian approach above, including the following functions:

```
def orientedFilterMagnitude(im):
    ...
    return mag, theta
```

Computes the boundary magnitude and orientation using a set of oriented filters, such as elongated Gaussian derivative filters. Explain your choice of filters. Use at least four orientations. One way to combine filter responses is to compute a boundary score for each filter (simply by filtering with it) and then use the max and argmax over filter responses to compute the magnitude and orientation for each pixel.

```
def edgeOrientedFilters(im):
    ...
    return bmap
```

Similar to part **(a)**, this should call `orientedFilterMagnitude`, perform the non-maxima suppression, and output the final soft edge map.

Evaluation: The provided folder *evaluateBSDS500* will evaluate your boundary detectors against the ground truth segmentations and summarize the performance. You can run the evaluation attempt using the folder *bench* to check if the setup is successful. You will need to edit to put in your own directories and edge detection functions for your final answer.

Write-up: Include your code with your electronic submission. In your write-up, include:

- Description of any design choices and parameters;
- The bank of filters used for part **(b);**
- **Qualitative results**: choose two example images; show input images and outputs of each edge detector;
- **Quantitative results**: Tables showing the overall F-score and the average F-score (which is outputted as text after running the evaluation script).

Problem 3 Feature Tracker (40 points)

You can only use the following Python libraries: OpenCV, numpy, math, and matplotlib.

Submission List:

- Codes for the implementation and displaying results in part **A** and **B** in the folder *solutions*.
- All code files in the folder *prob_3*.
- Display the results and write explanations (pseudocodes are needed), for part **A** and **B** in the *your_school_id_name.pdf*. (*e.g., 1190190xx_San_Zhang.pdf*).

For this problem, you will track features from the image sequence *hotel.seq0.png*, ..., *hotel.seq50.png* in the folder *images*. Since this is a two part problem, we have included precomputed intermediate results in the supplemental material in case you are unable to complete any portion.

Please also include pseudocode in your report. Furthermore, do not use existing keypoint detectors, trackers, or structure from motion code, such as found in OpenCV.

A. Keypoint Selection

For the first frame, use the second moment matrix to locate strong corners to use as keypoints. These points will be tracked throughout the sequence.

You can either use the Harris criteria (1), *or* the Shi-Tomasi/Kanade-Tomasi criteria (2). Here M is the second moment matrix, λ_1, λ_2 are the eigenvalues of M, and τ is the threshold for selecting keypoints:

$$\det(M) - \alpha \cdot \text{trace}(M)^2 \geq \tau \quad (1)$$

$$\min(\lambda_1, \lambda_2) \geq \tau \quad (2)$$

If using the Harris criteria, it is good to choose $\alpha \in [0.01, 0.06]$. Choose τ so that edges and noisy patches are ignored. Do local non-maxima suppression over a 5×5 window centered at each point. This should give several hundred good points to track.

Required output:

Display the first frame of the sequence overlaid with the detected keypoints. Ensure that they are clearly visible.

Suggested Structure:

Write this as a function such as

```
def getKeypoints(im, tau):
    ...
    return keyXs, keyYs
```

Be sure to smooth the gradients when constructing the second moment matrix.

References:

1. Harris and M. Stephens. A Combined Corner and Edge Detector. 1988
2. J. Shi and C. Tomasi. Good Features to Track. 1993

B. Tracking

Apply the Kanade-Lucas-Tomasi tracking procedure to track the keypoints found in part A. For each keypoint *k*, compute the expected translation from $(x, y) \rightarrow (x', y')$:

$$I(x', y', t+1) = I(x, y, t) \tag{3}$$

This can be done by iteratively applying (4): Given the i^{th} estimate (x'_i, y'_i), we want to update our estimate $(x'_{i+1}, y'_{i+1}) = (x'_i, y'_i) + (u, v)$. Here, W is a 15×15 pixel window surrounding the keypoint, which is located at (x, y) in frame t. I_x, I_y are the x, y gradients of image $I(x, y, t)$, computed at each element of W at time t. $I_t = I(x', y', t+1) - I(x, y, t)$ is the "temporal" gradient. A fixed, small number of iterations should be sufficient.

$$\begin{aligned} (x'_0, y'_0) &= (x, y) \\ I_t &\approx I(x'_i, y'_i, t+1) - I(x, y, t) \end{aligned}$$

$$\begin{bmatrix} \sum_W I_x I_x & \sum_W I_x I_y \\ \sum_W I_x I_y & \sum_W I_y I_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} \sum_W I_x I_t \\ \sum_W I_y I_t \end{bmatrix} \tag{4}$$

$$(x'_{i+1}, y'_{i+1}) = (x'_i, y'_i) + (u, v)$$

This should be applied iteratively, that is, begin with $(x'_0, y'_0)^T = (x, y)^T$, which is needed to compute I_t. Use this I_t to estimate $(u, v)^T$, which can in turn be used to compute $(x'_1, y'_1) = (x'_0, y'_0) + (u, v)$, and so on. Note that $(x', y')^T$ (and $(x, y)^T$) need not be integer, so you will need to interpolate $I(x', y', t+1)$ (I_x, I_y, ..., *etc.*) at these non-integer values.

Some keypoints will move out of the image frame over the course of the sequence. Discard any track if the predicted translation falls outside the image frame.

Required Output:

1. For 20 random keypoints, draw the 2D path over the sequence of frames. That is, plot the progression of image coordinates for each of the 20 keypoints. Plot each of the paths on the same figure, overlaid on the first frame of the sequence.
2. On top of the first frame, plot the points which have moved out of frame at some point along the sequence.

Suggested Structure:

1. Compute new X, Y locations for all starting locations. Precompute gradients I_x, I_y here, then compute translation for each keypoint independently:

```
def predictTranslationAll(startXs, startYs, im0, im1):
  …
  return newXs, newYs
```

[newX newY] = predictTranslation(startX, startY, Ix, Iy, im0, im1);

2. For a single X, Y location, use the gradients `Ix`, `Iy`, and images `im0`, `im1` to compute the new location. Here it may be necessary to interpolate `Ix`, `Iy`, `im0`, `im1` if the corresponding locations are not integer:

```
def predictTranslation(startX, startY, Ix, Iy, im0, im1):
  …
  return newX, newY
```

References:

1. Carlo Tomasi and Takeo Kanade. Detection and Tracking of Point Features. 1992