Visual Navigation for Flying Robots
D. Cremers, X. Gao, V. Usenko
Summer Semester 2018

Computer Vision Group
Department of Informatics
Technical University of Munich

Exercise Sheet 4

Topic: Visual Odometry

Submission deadline: Sunday, 13.05.2018, 23:59 pm
Hand-in via email to visnav_ss2018@vision.in.tum.de

General Notice

The exercises should be done by yourself, but the final project should be done in teams of two to three students. We will use Ubuntu 16.04 in this lab course. It is already installed on the lab computers. If you want to use your own laptop, you will need to install Ubuntu by yourself.

Files related with the exercise will be placed in directories like 'paper/' or 'doc/'. Please read these materials before start answering the questions.

Exercise 1: Feature based visual odometry

In the lecture we've explained how to detect ORB features in the images. In this exercise you need to implement the ORB detection, descriptor computing, descriptor matching, and then use the matched points to estimate the camera motion. We provide you two RGB-D image pairs: ORB1.png, ORB2.png, ORB1_depth.png, ORB2_depth.png, please use these four images to complete the following tasks. The code framework is provided in computeORB.cpp.

1. ORB detector. The ORB key point is an oriented FAST, in which we need to compute the angle of each key point. We use OpenCV's FAST detector to get fast key points, but you need to implement the angle computation. The angle θ of a key point can be denoted as:

$$\theta = \arctan(m_{01}/m_{10}), \tag{1}$$

   where m_{01} and m_{10} are the moments defined in the slides. Here we choose a 16x16 patch around a key point, which means for key point at u, v, we take the patch from $(u-8, v-8)$ to $(u+7, v+7)$. Please implement the computeAngle() function using this notation.

   Hints: (i) Because we need a patch to compute the angle, key points that are too close to the image boundary should be removed. (ii) If you plot the angle of the key points, it will look like they are all pointing at the brighter part of the image (see Fig. 1). (iii) std::atan() and std::atan2() will give you a radian angle, but OpenCV uses the degree unit. So please convert it if needed.

Figure 1: Oriented FAST key points.

2. ORB descriptor. ORB uses BRIEF descriptor, which is a just 256 or 128 bits containing 0 and 1s. For each bit we need to compare the image intensity around the detected key point. The algorithm is described as below:

   - Given image I, key point (u, v) and its angle θ, let's take 256 bits as an example. The descriptor can be denoted as a vector

     $$\mathbf{d} = [d_1, d_2, \ldots, d_{256}], d_i = \{0, 1\}.$$

   - For each $i = 1, \ldots, 256$ d_i is computed as follows. Take two points around (u, v), say, $\mathbf{p}, \mathbf{q}$ (this is called as the ORB pattern), and rotate it according to θ:

     $$\begin{bmatrix} u_p{}' \\ v_p{}' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} u_p \\ v_p \end{bmatrix}. \tag{2}$$

     where u_p, v_p is the coefficients of $\mathbf{p}$ and same for $\mathbf{q}$. We denote the rotated points as $\mathbf{p}', \mathbf{q}'$, then we compare the image intensity of $I(\mathbf{p}')$ and $I(\mathbf{q}')$. If $I(\mathbf{p}')$, then $d_i = 0$, otherwise set $d_i = 1$.

   In order to simplify the coding, we use 256 boolean variables to represent the descriptor [1]. Please implement the computeORBDesc(). Note the $\mathbf{p}, \mathbf{q}$ (or ORB pattern) is given in the code, which is randomly chosen but we need to keep it same when computing the descriptors.

   Hints:

   - For $\mathbf{p}, \mathbf{q}$ we also need boundary checking. If the pattern goes outside the image, we set the descriptor vector to empty and ignore it when matching the key points.

[1] 32 bytes will be more compact but requires bit operator.

- Also please be careful about the degree and radian unit when calling the sin() and cos() functions.

3. Brute force matching of ORB features. After computing the descriptors, we need to match them according to the descriptors. Brute force matching is a simple and commonly used approach for feature matching, especially when the number of features is not large. Given two sets of descriptors, say, $\mathbf{P} = [p_1, \ldots, p_M]$ and $\mathbf{Q} = [q_1, \ldots, q_N]$, then for each point in $\mathbf{P}$, we find a point in $\mathbf{Q}$ that has the minimum (Hamming) distance. In practice, we also the a maximum distance threshold d_{max} and skip those key points whose distance is large than d_{max}. Please implement the bfMatch() function according to these statements, and we set $d_{max} = 50$ in the exercise.

   Hints: (i) You need to implement the Hamming distance computation. (ii) The cv::DMatch struct stores the matching result, where queryIdx is the key point index in image 1, and trainIdx is the index in image 2. (iii) The matching results should be same (or similar) as Fig. 2.

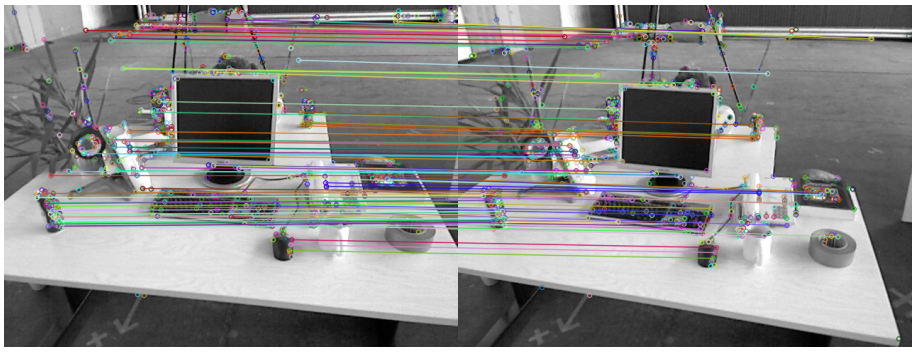

Figure 2: Matched key points.

4. Pose estimation. Using the matched key points, we can estimate the camera motion from image 1 to image 2. Since we provide RGB-D images, so we can use all the 2D-2D, 3D-2D, 3D-3D methods in this exercise. Please implement the poseEstimation function series according to the knowledge we talked about in the course, and compare the results of different approaches.

   Hints: (i) OpenCV provides some functions like findEssential(), solvePnP() and recoverPose() for pose computation, please find their documentation and use them if needed. (ii) In 3D-2D and 3D-3D cases we use the depth image to get the 3D points. If the depth value is missing (set to zero in the depth image), please just skip this point. (iii) Three approaches will give you similar (but not the same) results, and the 2D-2D estimated translation will have a scale difference with the other two.

Exercise 2: LK optical flow

Optical flow is also a very popular method to track the corner points. In this exercise we are going to implement a optical flow algorithm using the Gauss-Newton's framework. An optical flow survey paper is provided in [1], please read it if you have interest.

1. Single layer forward-additive optical flow. First we start from the simplest optical flow, namely the single layer forward-additive flow, and then expand it to inverse and multi-layer. We model the optical flow as a nonlinear optimization problem and solve it in a Gauss-Newton's way. We provide two images for this exercise: LK1.png and LK2.png, then we extract the GFTT corners [2] in image 1, and track them in image 2 using optical flow. Let the two images be I_i, I_2, the key points in image 1 are $\mathbf{P} = \{\mathbf{p}_i\}$ where $\mathbf{p}_i = [x_i, y_i]^{\mathrm{T}}$ is the pixel coordinate. Consider the i-th point, we want to compute it motion $\Delta x_i, \Delta y_i$:

$$\min_{\Delta x_i, \Delta y_i} \sum_{W} \left\| I_1\left(x_i, y_i\right) - I_2\left(x_i + \Delta x_i, y_i + \Delta y_i\right) \right\|_2^2, \tag{3}$$

   which means minimizing the quadratic pixel error and $\sum_{W}$ means we assume the pixel values in this window W don't change. In practice we choose an 8×8 window, namely from $(x_i - 4, y_i - 4)$ to $(x_i + 3, y_i + 3)$. Obviously, this is a forward-addtive optical flow, and the above least squares problem can be solved by Gauss-Newton iteration. Please answer the following questions and based on your answers, implement the function OpticalFlowSingleLevel in the optical_flow.cpp file.

   - How to derive the jacobian of the error related to the motion?

   Hints: (i) Same as the previous job, you still need to remove the points that were placed near the image boundary, otherwise your image blocks may go outside the border. (ii) This function is called a single-layer optical flow. After this we will implement the multilayer optical flow based on this function. In the main function, we test single-layer optical flow and multi-layer optical flow for two images and compare them with OpenCV results. As a verification, the forward single-layer optical flow results should be similar as the results shown in Fig. 3. The result is not very good, but most of the key points are still correctly tracked.

2. Inverse-additive optical flow. After you implement the above algorithm, you will find that at the beginning of the iteration, the Gauss-Newton's calculation depends on the gradient information of I_2 at (x_i, y_i). However, the corner extraction algorithm only guarantees that $I_1(x_i, y_i)$ is a corner point (which means there is a large gradient here), but for I_2, we have no way to assume that I_2 in x_i, y_i also has a large gradient, so the Gauss-Newton is not necessarily true. The inverse optical flow does a clever trick by replacing the original $I_2(x_i + \Delta x_i, y_i + \Delta y_i)$ with the gradient at $I_1(x_i, y_i)$. The benefits are:

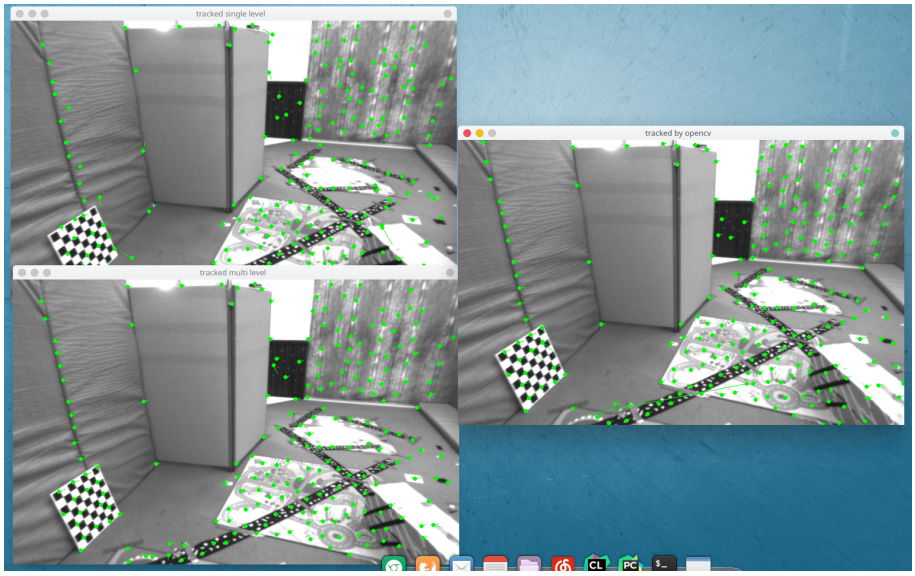

Figure 3: Results of optical flow. The multi-layer result should be similar as OpenCV.

- $I_1(x_i, y_i)$ is the corner point, the gradient is always meaningful.
- The gradient at$I_1(x_i, y_i)$ does not change with iterations, so it only needs to be calculated once and it can be used in subsequent iterations, saving a lot of computing time.

We add a boolean parameter to the OpticalFlowSingleLevel function to specify whether we want to use a normal algorithm or a inverse algorithm. Please follow the above instructions to complete the reverse LK optical flow method.

3. Extend to multi layer. Through the experiments, we can see that the optical flow method can only estimate the error within a few pixels. If the initial estimate is not good enough, or the image motion is too large, the optical flow method cannot be effectively estimated (unlike feature matching). However, using an image pyramid can make the optical flow less sensitive to image motion. Next, please use a 4 level image pyramid with a zoom factor of 2, to implement a coarse-to-fine LK optical flow. The function is in OpticalFlowMultiLevel().

   After completing the implementation, give your optical flow screenshots (forward, reverse, multi level forward, multi level reverse). Then answer the following questions:

   - What does the "coarse-to-fine" mean in multi level processing?
   - What is the difference between pyramid use in the optical flow method and pyramid in the feature method?

   Hints: You can use the single layer optical flow written before to help you achieve multi-layer optical flow.

Exercise 3: Direct method

We say that the direct method is an intuitive extension of optical flow. In the optical flow, we estimate the translation of each pixel (in the case of additive flow). In the direct method, we minimize the brightness error to estimate the camera's rotation and translation (in the form of Lie algebra). Now we will use a very similar approach to the previous one to implement the direct method. You can feel the close relationship between the direct method and the optical flow.

1. Single layer direct method. In this exercise, you will use some of the images in the Kitti dataset. Given left.png and disparity.png, we know that we can get 3D information of any point in left.png through these two images. Now, please use the direct method to estimate the pose of the images 000001.png to 000005.png. We call left.png as **reference image**, and any one of the images 000001.png -000005.png is **current image**, as shown in Figure 4 shown. Set the target to be evaluated as $\mathbf{T}_{\text{cur,ref}}$, then take a set of points $\{\mathbf{p}_i\}$ in ref. The pose can be solved by minimizing the following objective function:

$$E(\mathbf{T}_{\text{cur,ref}}) = \frac{1}{N} \sum_{i=1}^{N} \sum_{W_i} \left\| I_{\text{ref}} \left(\pi(\mathbf{p}_i) \right) - I_{\text{cur}} \left(\pi \left(\mathbf{T}_{\text{cur,ref}} \ \mathbf{p}_i \right) \right) \right\|_2^2, \tag{4}$$

   where N is the number of points, the π function is the projection function of the pinhole camera $\mathbb{R}^3 \mapsto \mathbb{R}^2$, and W_i is a small window around the point i. With the optical flow method, this problem can be solved by the Gauss-Newton method. Please answer the following questions and then implement the DirectPoseEstimationSingleLayer function in direct_method.cpp.

   - What is the Jacobian dimension of the error relative to the estimated variable?
   - What is the size of the window? Can I take a single point?

   Here are some hints in the implementation process:

   - This time we randomly take 1000 points in the reference image, not the corner points. Please consider why direct methods can work without taking corner points.
   - Due to the camera motion, points in the reference image may go beyond the subsequent image boundaries after being projected. So the final objective function needs to average the points that are projected internally, instead of averaging all the points. In the program, we mark the points projected inside with good points.
   - The single-layer direct method doesn't work very well, but you can see that the objective function of each iteration will decrease.

Figure 4: Images used in this exercise.

2. Multi layer direct method. In the following, similar to the optical flow, we can also extend the direct method to a multi-level pyramid in a coarse-to-fine process. The direct method of multilayer pyramids allows the image to track all points even with large movements. Here we use a four-level pyramid with a zoom factor of 2 to implement the direct method on the pyramid. Please implement the DirectPoseEstimationMultiLayer function. Here are some tips:

   - When scaling an image, the camera intrinsics also need to change. So, for example, if the image is doubled, how should f_x, f_y, c_x, c_y be changed?
   - According to the coarse-to-fine process, the pose estimation result of the upper layer image can be used as the initial condition of the next layer image.
   - During debugging, you can draw a projection of each point on reference and current to see if they correspond. If they correspond exactly, the position and pose estimates are accurate.

   As a verification, the pose shift portion of images 000001 and 000005 should be close to:

$$\begin{aligned} \mathbf{t}_1 &= [0.005876, -0.01024, -0.0725]^T \\ \mathbf{t}_5 &= [0.0394, -0.0592, -3.9907]^T \end{aligned} \tag{5}$$

It can be seen that the vehicle is basically straight forward.

3. Discussion. You have now implemented the Gauss-Newton direct method on the pyramid. You can adjust some parameters in the experiment, such as the number of image points, the size of the patch around each point, and so on. Please consider the following questions:

   - In the direct method, can we use similar concepts in optical flow and propose the concept of inverse, compositional? Do they make sense?
   - Consider where the above algorithm can be cached or accelerated?
   - Why can we take random points instead of extracting corners?
   - Please summarize the similarities, differences, advantages and disadvantages of the direct method with respect to the feature point method.

Submission instructions

A complete submission consists both of a PDF file with the solutions/answers to the questions on the exercise sheet and a ZIP file containing the source code that you used to solve the given problems. Note all names of your team members in the PDF file. Make sure that your ZIP file contains all files necessary to compile and run your code, but it should not contain any build files or binaries. Please submit your solution via email to `visnav_ss2018@vision.in.tum.de`.

References

[1] S. Baker and I. Matthews, "Lucas-kanade 20 years on: A unifying framework," *International journal of computer vision*, vol. 56, no. 3, pp. 221–255, 2004.

[2] J. Shi and C. Tomasi, "Good features to track," in *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR'94., 1994 IEEE Computer Society Conference on*, pp. 593–600, IEEE, 1994.