Division of Engineering Science
UNIVERSITY OF TORONTO

ROB501: Computer Vision for Robotics

Project #2: Camera Pose Estimation

Fall 2019

Overview

Camera pose estimation is a very common task for robotic vision—we typically wish to know the pose (position and orientation) of the camera in the environment at all times. The general problem of pose estimation can be tricky (because you need to know something about scene geometry). In this project, you will learn how to estimate the pose of the camera relative to a *known* object, in this case a planar checkerboard camera calibration target of fixed size. The goals are to:

- provide practical experience with image smoothing and subpixel feature extraction, and
- assist in understanding the nonlinear least squares optimization method.

The due date for project submission is **Thursday, October 17, 2019, by 11:59 p.m. EDT**. All submissions will be in Python 3 via Autolab (more details will be provided in class and on Quercus); you may submit as many times as you wish until the deadline. To complete the project, you will need to review some material that goes beyond that discussed in the lectures—more details are provided below.

Details

For problems in which the pose of the camera must be determined with a high degree of accuracy (e.g., camera tracking for high fidelity 3D reconstruction), it is not unusual to insert a known calibration object into the scene; knowledge of the geometry of the calibration object can then be used to assist in pose estimation (assuming the object is visible).

As part of this project, you will estimate the camera pose relative to a checkerboard target whose squares are **63.5 mm** in size (on a side). You may assume that the checkerboard is perfectly flat, that is, the z coordinate of any point lying on the board is exactly zero (in the frame of the target). Sample images are shown at the bottom of the page. You may also assume that each image has already been unwarped to remove any lens distortion effects (you will be estimating pose parameters only; the intrinsic parameters are also assumed to be known already).

Please clearly comment your code and ensure that you only make use of the Python modules and functions listed at the top of the code templates. We will view and run your code.

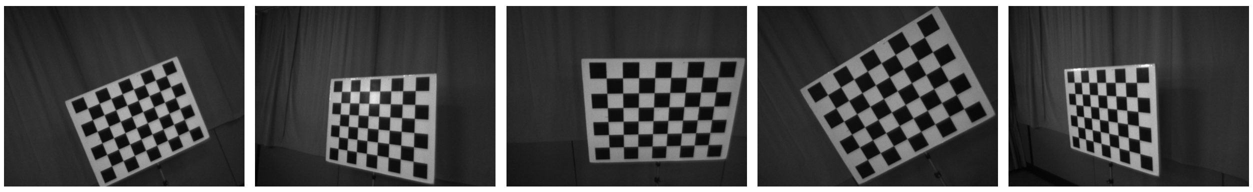
(a) (b) (c) (d) (e)

Figure 1: Sample images for use in testing your pose estimation algorithm.

Part 1: Image Smoothing and Subpixel Feature Extraction

To determine the camera pose, you will need to carefully extract a set of known feature points from an image of the target. Correspondences between the observed 2D image coordinates and the known 3D coordinates of the feature points (landmarks) then allows the pose to be determined (see Parts 4). Typically, for a planar checkerboard target, *cross-junction* points are used, for two reasons, 1) they are easy to extract, and 2) they are *invariant* to perspective transformations. A cross-junction is defined as the (ideal) point at which the diagonal black and white squares meet. In the sample images, the number of cross-junctions is $8 \times 6 = 48$.

There are variety of ways to identify the cross-junctions (for example, using the Harris corner detector). Usually, the coarse estimates of the cross-junction positions in each image are then refined using a saddle point detector, such as the one described in the following paper (included in the project archive):

> L. Lucchese and S. K. Mitra, "Using Saddle Points for Subpixel Feature Detection in Camera Calibration Targets," in *Proceedings of the Asia-Pacific Conference on Circuits and Systems (APCCAS)*, vol. 2, (Singapore), pp. 191–195, December 2002.

The saddle point is the best subpixel estimate of the true position of the cross-junction. Images are typically smoothed with a Gaussian filter prior to computing the saddle points—this is done because, in many cases, the cross-junctions are not clearly defined (due to, e.g., image quantization errors). If you zoom in on one of the sample images, you may notice this effect.

Your first task is to write a Python function that computes the position of the saddle point in a small image patch. This can be carried out by fitting a hyperbolic paraboloid to the smoothed intensity surface. The relevant fitting problem is defined by Eqn. (4) in the Lucchese paper, and is solved (unsurprisingly) using linear least squares! For this portion of the assignment, you should submit:

- A single function, `saddle_point.py`, which accepts a small image patch as input and attempts to find a saddle point using the algorithm described in the paper. Note that the image coordinates of the saddle point should be returned with subpixel precision (i.e., as doubles).

You will have access to the SciPy `gaussian_filter` function, which will perform image blurring (smoothing with a symmetric Gaussian kernel of fixed standard deviation)—feel free to try it out! For convenience, all of the Autolab tests for Part 1 use patches that have been pre-blurred in advance.

Note that two steps are required to find the saddle point: Eqn. (4) must be solved *first*, for the parameters α, β, γ, δ, ϵ, and ζ; the coordinates of the saddle point are then found using the next equation in the paper.

Part 2: Extracting All Cross-Junctions

Your second task is to implement a Python function that returns an ordered list (row-major, from top left) of the cross-junction points on the target, to subpixel precision as determined by the saddle point detector. In every case, we will provide a bounding polygon that encloses the target (where the bounding polygon has points ordered clockwise from upper left); the upper left cross-junction should be taken as the origin of the target frame, with the x-axis pointing to the right in the image and the z-axis extending into the page. Using this information, you will be able compute the metric coordinates of each (ideal) cross-junction (in metres)—we have also provided a `.mat` file that contains this set of 3D coordinates. For this portion of the assignment, you should submit:

- A single function, `cross_junctions.py`, which accepts an image and a bounding polygon, and extracts all of the cross-junctions on the planar target and returns their coordinates with subpixel precision (in row-major order). The function should also accept the 3D (world) coordinates of the points (landmarks); the number of image features you extract should be the same as the number of landmarks.

对比验证的

The first set of (x, y) bounding polygon coordinates will *always* be closest to the upper leftmost cross-junction on the target (this should make ordering easier). Note that you may need to copy and paste the 'innards' of your `saddle_point.py` function into the appropriate section of the `cross_junctions.py` function. You should develop a novel way to coarsely localize the cross-junctions, followed by subpixel refinement with the saddle point detector. *Note that this part of the project may require substantial effort.*

Part 3: Camera Pose Jacobians

Upon completing Parts 1 and 2 of the project, you should have a function, (`cross_junctions.py`), that produces a series of 2D-3D feature correspondences that can be used for pose estimation. You will implement pose estimation using a nonlinear least squares (NLS) procedure that incorporates all of the available data.

The solution to the **PnP problem** (i.e., Perspective-n-Points) is described in Section 6.2 of the Szeliski text. Although there is a linear algorithm for the problem, it does not work when all points are coplanar. Instead, we will provide you with an initial guess for the camera pose ($\pm 10^\circ$ and 20 cm, approximately). You will need to know the camera intrinsic calibration matrix, which in this case is

$$\mathbf{K} = \begin{bmatrix} 564.9 & 0 & 337.3 \\ 0 & 564.3 & 226.5 \\ 0 & 0 & 1 \end{bmatrix},$$

where the focal lengths and principal point values are in pixels. To make things easier, we will use an *Euler angle* parameterization for the camera orientation. It will be necessary to solve for six parameters: the x, y, and z camera translation, and the roll, pitch, and yaw Euler angles that define the camera orientation. **Note that we wish to solve for the pose of the camera relative to the target,** $\mathbf{T}_{WC}$ (i.e., a 4×4 homogeneous pose matrix). When expressed in terms of the roll (ϕ), pitch (θ), and yaw (ψ) Euler angles, the rotation matrix that defines the orientation of the camera frame relative to the world (target) frame is

$$\begin{aligned} \mathbf{C}_{WC}(\psi, \theta, \phi) &= \mathbf{C}_(\psi)\, \mathbf{C}_(\theta)\, \mathbf{C}_(\phi) && (1) \\ &= \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} && (2) \\ &= \begin{bmatrix} \cos\psi\cos\theta & \cos\psi\sin\theta\sin\phi - \sin\psi\cos\phi & \cos\psi\sin\theta\cos\phi + \sin\psi\sin\phi \\ \sin\psi\cos\theta & \sin\psi\sin\theta\sin\phi + \cos\psi\cos\phi & \sin\psi\sin\theta\cos\phi - \cos\psi\sin\phi \\ -\sin\theta & \cos\theta\sin\phi & \cos\theta\cos\phi \end{bmatrix}. && (3) \end{aligned}$$

In order to compute the NLS solution, you will need the (full) Jacobian matrix for the image plane points (observations) with respect to the (six) pose parameters. The full Jacobian is composed of a series of 2×6 sub-matrices, stacked vertically. For this portion of the assignment, you should submit:

- A single function, `find_jacobian.py`, that computes a 2×6 Jacobian matrix for each image plane (feature) observation with respect to the camera pose parameters.

We will use the pinhole projection model, where the image plane coordinates of the projection of landmark j, with 3D position $\bar{\mathbf{p}}_j$, are

$$\tilde{\mathbf{x}}_{ij} = \tilde{\mathbf{K}} \left(\mathbf{T}_{WC_i}\right)^{-1} \bar{\mathbf{p}}_j \qquad (4)$$

for camera pose i. Two helper functions to convert between Euler angles and rotation matrices are available on Autolab (and in the code package that accompanies this project document) to assist you.

Part 4: Camera Pose Estimation

The final step is to set up, and then solve, the nonlinear system of equations for pose estimation, using nonlinear least squares. For this portion of the assignment, you should submit:

- A function, `pose_estimate_nls.py`, which accepts an intrinsic calibration matrix, a set of 2D-3D correspondences (image-target) and an initial guess for the camera pose, and performs a nonlinear least squares optimization procedure to compute an updated, optimal estimate of the camera pose.

For testing, you may use the example image included in the project archive.

Grading

Points for each portion of the project will be assigned as follows:

- Saddle point function – **12 points** (4 tests $\times$ 3 points per test)

  Each test uses a different (pre-blurred) image patch containing a cross-junction. The estimate of the (subpixel) position of the saddle point must be within 1.0 pixels of the reference position to pass.

- Cross-junction extraction – **15 points** (3 tests $\times$ 5 points per test)

  Each test uses a different image (of the calibration target); to pass, the extraction function must return the full set of image plane points (48), and the average position error (over all cross-junctions) must be less than 2 pixels.

- Jacobian function – **11 points** (2 tests; 6 points and 5 points)

  The computed Jacobian matrix (for each test) is checked for accuracy—there is an exact solution for each camera pose and landmark point.

- NLS pose estimation function – **12 points** (3 tests $\times$ 4 points per test)

  There are three tests, each of which uses a different (holdout) image of the calibration target. The returned, optimized pose solution must be 'close' to the reference solution (within about 5 cm and 2°).

Total: **50 points**

Grading criteria include: correctness and succinctness of the implementation of support functions, proper overall program operation, and code commenting. Please note that we will test your code *and it must run successfully*. Code that is not properly commented or that looks like 'spaghetti' may result in an overall deduction of up to 10%.