Division of Engineering Science
UNIVERSITY OF TORONTO

ROB501: Computer Vision for Robotics

Project #4: Stereo Visual Odometry

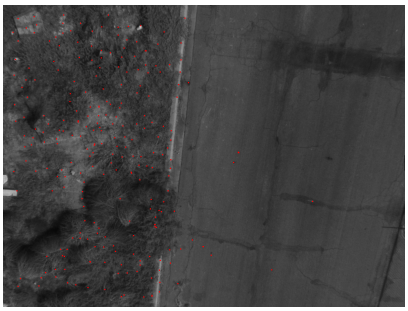
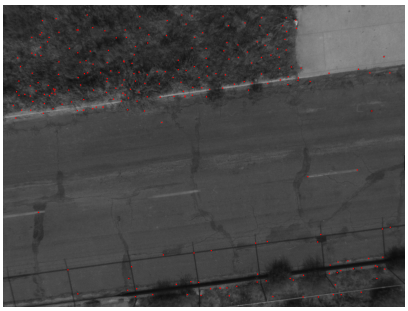
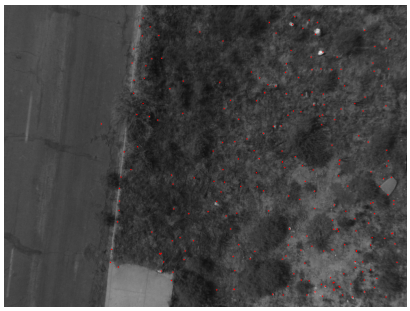
Fall 2019

Overview

Stereo vision is highly versatile—it can be used for motion estimation (a.k.a. *visual odometry*) among other tasks. In a basic stereo VO system, a set of 3D landmark points are triangulated (based on the stereo rig geometry) from corresponding sparse image features in the left and right cameras; landmarks are then matched (temporally) across stereo pairs, and the two sets of 3D points are aligned to determine the incremental change in pose between image acquisitions. In this project, you will implement portions of a stereo VO pipeline, and (later) test on data gathered by an aerial robot platform. The goals are to:

- expand on your understanding of the use of stereo vision, in particular for motion estimation tasks, and
- introduce the visual odometry algorithm developed for the Mars Exploration Rovers!

The due date for project submission is **Friday, November 29, 2019, by 11:59 p.m. EST**. 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. The project has three (or four) parts, worth a total of **50 points** (plus bonus points).

To ease your experimentation with stereo VO, we will provide a dataset of 320 stereo pairs acquired during the flight of an autonomous helicopter at a test site in Los Angeles, California (see samples above). The stereo rig looked approximately downward (nadir view); illumination conditions were ideal during data collection, and there was ample texture in the images for feature tracking.

The full dataset, which will available from a Dropbox link (to be posted on Quercus), will be used for the bonus portion of the assignment. The dataset file is a large (530 MB) — when unzipped, it will include four subdirectories: `camera/` contains a Python function that loads the stereo camera parameters (pinhole models); `images/` contains unwarped and rectified left and right stereo images; `tracks/` contains left stereo images with tracked feature points already identified (more on this below); `matches/` contains `.mat` files specifying both spatial (intra-pair) and temporal (inter-pair) matches between left and right images, and left images, respectively, in sequence. As a reference, feature tracking was performed using the KLT algorithm and left-right matches were determined using normalized cross-correlation (with subpixel fitting).

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.

Part 1: Stereo Triangulation

The first step in stereo visual odometry is to triangulate the 3D positions of a series of point landmarks relative to the current stereo camera reference frame. The 3D landmark positions can be determined from image feature observations in the left and right stereo images. As discussed in the lectures, the complication, in this case, vis-à-vis Project #3, is that the feature positions on the left and right image planes do not have identical vertical coordinates. Thus, rays back-projected through the cameras will not intersect at a single point in space; the situation is illustrated on Slide 5 of Lecture 11.

Given that the rays do not exactly intersect, an alternative is to find the shortest line segment connecting two points on the rays, and to treat the midpoint of this segment as the estimated landmark position (thankfully, we only need to deal with two rays, and so do not need to consider the least squares solution from Lecture 11). The relevant calculation details are given in the paper:

> Y. Cheng, M. W. Maimone, and L. H. Matthies, "Visual Odometry on the Mars Exploration Rovers," in *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, vol. 1, Big Island, Hawaii, USA, Oct. 2005, pp. 903–□□910.

See Section 2 of the paper for more information. For this portion of the project, you should submit:

- A function, `triangulate.py`, that accepts left and right camera intrinsic calibration matrices, left and right camera pose matrices (in the world frame), and left and right image plane points, and computes the 3D position of a triangulated landmark point.

In order to compute the ray endpoints, you will require a unit vector pointing along each ray (designated as r_1 and r_2 in the paper). For the left camera, this unit vector can be determined from the observed image plane coordinates (u_L, v_L) as

$$\mathbf{r}_L = \mathbf{C}_{WC_L}\,\mathbf{K}_L^{-1}\begin{bmatrix} u_L \\ v_L \\ 1 \end{bmatrix}, \qquad \hat{\mathbf{r}}_L = \frac{\mathbf{r}_L}{\|\mathbf{r}_L\|},$$

and analogously for the right camera.

For each landmark point, you will also need to determine the 3D uncertainty ellipsoid associated with its estimated position. This calculation can be carried out inside the `triangulate.py` function—the majority has already been implemented for you. However, you will *still need to compute an image plane Jacobian*, that is, the Jacobian of the ray direction with respect to the image plane coordinates (denoted as r' in the paper). Thankfully this is not as difficult as in Project 2!

Given the image plane measurement covariance matrices (which are supplied for you), it is possible to determine the uncertainty associated with each landmark position (in 3D) using *first order uncertainty (or error) propagation*—this is covered in detail on Page 3 of the paper (see Eqn. 11). First order error propagation uses a Jacobian matrix to propagate the image plane uncertainties (assumed to be Gaussian distributions on the image plane) 'into' 3D. The relevant formula is:

$$\mathbf{\Sigma}_P = \mathbf{J}_P \begin{bmatrix} \mathbf{\Sigma}_L & \mathbf{0}_{2\times 2} \\ \mathbf{0}_{2\times 2} & \mathbf{\Sigma}_R \end{bmatrix} \mathbf{J}_P^T.$$

This is a slightly different use for Jacobian matrices, but the method of computation is exactly the same as the method you have employed previously. We have used the notation $\mathbf{J}_P$ for the Jacobian matrix (to be consistent with the course lectures); this matrix is identified as P' in the Cheng paper. Also note that covariance matrices are represented by the $\mathbf{\Sigma}$ symbol above (the capital Greek letter 'sigma').

Part 2: Motion Estimation – Linear Methods

The most straightforward (and fastest) way to compute the rigid body transform between two point clouds (when correspondences are known) is to make use of a linear method (linear least squares, for example). Singular value decomposition can, in fact, be used to solve linear least squares problems—the Wikipedia article on SVD does quite a good job of explaining why this is the case (see the section on "Intuitive interpretations" for more).

The paper by Cheng et al. gives some background on the linear approach, but does not specifically mention SVD. A full algorithm is given by Matthies in his PhD thesis (see Appendix B and Page 150) that is provided for you. It was the work in Matthies' thesis that formed the basis for the visual odometry software that ran on board the Mars Exploration Rovers (Spirit and Opportunity), as well as for DIMES.

The use of SVD to compute rigid body motion requires several 'tricks' to arrive at the formal algorithm. However, an important part of the algorithm (the part that we care about) is the weighting of each pair of points (before and after motion). The weights are given by Equation 14 in the Cheng paper: each weight is the reciprocal of the sum of the determinants of the 3D landmark covariance matrices. The result is a single weight value, which discards the geometric information in the covariance matrices but considers the overall 'quality' of each observation. Note that the determinant of a covariance matrix is a measure of the 'volume' of the error ellipsoid (for a Gaussian distribution).

For this portion of the project, you should submit:

- A function, `estimate_motion_points_ls.m`, that accepts a set of pairs of corresponding 3D landmark points before and after some incremental motion, and their associated 3×3 covariance matrices, and computes the linear motion solution. You are only required to fill in the weighting computation (but be sure to look at how the other code works, the idea is to give you an idea of how each step is implemented!).

Part 3: Motion Estimation – Going Nonlinear

After completing Part 2, you should be able to initialize the motion estimation process (with a reasonable linear guess). Your next task is to implement an iterative nonlinear least squares point cloud alignment routine.

There is one difference in the nonlinear least squares algorithm to be implemented here, compared to the nonlinear least squares problems discussed in our early lectures. We wish to *weight* each observation by the inverse of its uncertainty, taking advantage of the structure (shape) of covariance ellipsoid. If $\mathbf{\Sigma}_i$ is the covariance matrix for one observation (pair of corresponding landmarks), the nonlinear update step can be written as

$$\underbrace{\left[\sum_i \mathbf{J}_i^T \mathbf{\Sigma}_i^{-1} \mathbf{J}_i\right]}_{\mathbf{A}} \Delta\mathbf{p} = \underbrace{\left[\sum_i \mathbf{J}_i^T \mathbf{\Sigma}_i^{-1} \mathbf{r}_i\right]}_{\mathbf{B}}.$$

for the vector of small parameter changes $\Delta\mathbf{p}$. Details on the full calculation (which varies slightly from the generic NLS update given above) are available in the Matthies' PhD thesis (again in Appendix B). Note that the 'weight' assigned to each observation is exactly $\mathbf{\Sigma}_i^{-1}$ (i.e., the inverse of the combined landmark position uncertainty). You do not have to follow his implementation exactly (e.g., you can reuse much of your camera pose estimation code), however you should pay particular attention to the covariance computation on Slide 18 of Lecture 16 (the $\mathbf{GRG}^T$ calculatioin is already part of `triangulate.py`).

For this portion of the project, you should submit:

- A function, `estimate_motion_ils.py`, that accepts a set of pairs of corresponding 3D landmark points before and after some incremental motion, and their associated 3×3 covariance matrices (produced by triangulation), and computes the nonlinear motion solution.

Specific pointers are given in the `estimate_motion_ils.py` template file, and a large part of the skeleton has been provided. Overall, only handful of lines of code need to be written (so don't write too much!). The complete set of equations that are can be used are given on pages 150–151 (bottom and top) of Matthies' PhD thesis (if you choose to go this route).

Bonus: The Stereo Visual Odometry Pipeline End-to-End

Before trying out this portion of the project, pat yourself on the back—you just finished writing chunks of software that have been running on Mars!

With Parts 1–3 finished, you've completed a good portion of the code required to run a full VO pipeline. One week before the due date, we will release additional software components and the helicopter dataset; you will be able to add your working pieces to produce a full VO solution. This will enable you to compare your compounded motion estimate with our reference trajectory. A small (10%) bonus will be awarded for completing this *optional* portion. The visualization should also be neat!

Grading

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

- Stereo triangulation function – **20 points** (4 tests $\times$ 5 points per test)

  The stereo triangulation function should compute the positions of the segment endpoints and the 3D landmark point with millimetre accuracy (assuming the baseline is in metres). We will also check the value of each entry in the 3×3 covariance matrix; each entry must differ from the reference solution by less than 1×10^{-6}.

- Linear motion estimation function – **6 points** (3 tests $\times$ 2 points per test)

  We will check the accuracy of both the rotation matrix and translation vector; each entry must differ from the reference solution by less than 1×10^{-9}. This should be possible because the solution is exact (and you are only implementing a small amount of additional code).

- Nonlinear motion estimation function – **24 points** (4 tests $\times$ 6 points per test)

  We will check the accuracy of both the rotation matrix and translation vector; each entry must differ from the reference solution by less than 1×10^{-9}.

Total: **50 points**

Bonus: Additional bonus points (5) will be awarded for implementing the complete VO solution using our framework; further details will be available on Quercus one week prior to the project due date.

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%.