TTK4255 Robotic Vision, Spring 2020

Exercise 4: Non-Linear Least Squares

Authored by Simen Haugo (simen.haugo@ntnu.no)

Due: See Blackboard

This exercise is a mix of programming problems and theory questions. You are free to use any programming language, but solutions will be provided in Python and Matlab, and we may not be able to help you effectively if you use a different language. We therefore recommend that you use either Python or Matlab. You should also do exercise 0 to get familiar with language features and libraries that are useful for the exercises.

Instructions

This exercise is **approved** / **not approved**. To get your exercise approved, submit your answers to the questions as a **PDF** to Blackboard. You **don't** have to submit your code. Feel free to use LaTeX, Word, handwritten scans, or any other tool to write your answers.

Getting help

If you have questions about the exercise, you can:

- Post a question on Piazza (you can post anonymously if you want)
- Ask for help during the tutorial sessions (see Piazza for time and place)

Relevant resources

Non-linear least squares optimization will be familiar to those who have taken TTK4135. For those who haven't taken this course (or are taking it right now), you should read the relevant chapters in Nocedal and Wright.

- Lecture 5.
- *Numerical Optimization* by Nocedal and Wright.
  - Ch. 2: Fundamentals of Unconstrained Optimization
  - Ch. 10: Least-Squares Problems (p. 245-256)

    Sections after "The Gauss-Newton Method" can be skipped. The book can be downloaded for free on SpringerLink when connected to Eduroam: https://link.springer.com/book/10.1007/978-0-387-40065-5

Provided code

The zip includes template Python and Matlab code that takes care of reading in data and generating the required output figures. Your job is to finish the implementation of the stub functions.

Background

Recall the Quanser 3-DoF helicopter from Exercise 1. In this exercise you'll implement an algorithm to track the helicopter's yaw, pitch and roll from fiducial markers. Compared with the DLT, this algorithm is more flexible and can easily incorporate constraints or other knowledge, such as kinematics, dynamics and information from other sensors. [1]

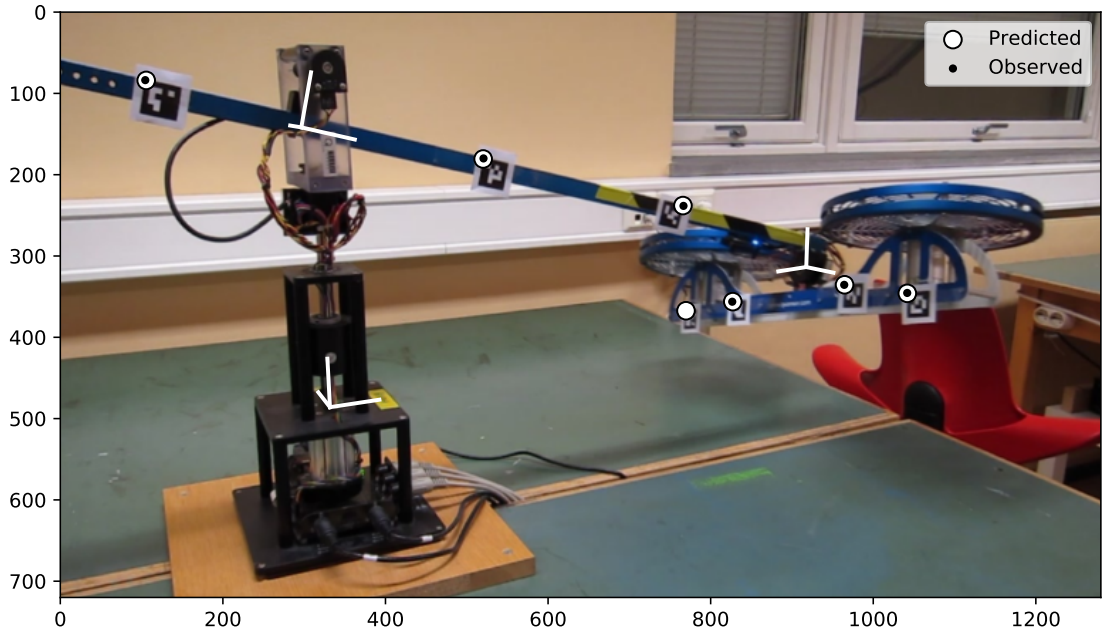

Figure 1: Coordinate frames and predicted versus measured marker coordinates.

As you learned in Exercise 1, the helicopter model relates points in the helicopter's coordinate frames to points in the image. Given the true yaw, pitch and roll, and the camera pose, the points in the helicopter should project to the same image coordinate as the corresponding detected marker. Otherwise, there is a *reprojection error* which can be minimized using numerical optimization to obtain the optimal parameters.

This is commonly posed as a non-linear least squares problem: Given m correspondences between known points in the object and the corresponding observed image coordinates, we can estimate the unknown parameters by minimizing the sum of squared reprojection errors:

$$f(\boldsymbol{\theta}) = \sum_{i=1}^{m} ||\hat{\boldsymbol{u}}_i(\boldsymbol{\theta}) - \boldsymbol{u}_i||^2 \tag{1}$$

Here, f is called the *objective function* and is usually interpreted as the cost, error or energy that needs to be optimized. $\boldsymbol{\theta} = (\psi, \theta, \phi)$ contains the unknown parameters that we want to estimate. $\hat{\boldsymbol{u}}_i(\boldsymbol{\theta})$ is the predicted pixel coordinate of marker i and $\boldsymbol{u}_i$ is the corresponding observed pixel coordinate. Finally, error is measured using the Euclidean distance: $||\boldsymbol{e}|| = \sqrt{\boldsymbol{e}^T \boldsymbol{e}}$.

Reminder: The Gauss-Newton method

A popular algorithm for solving non-linear least squares problems is the Gauss-Newton method. It starts at an initial estimate $\boldsymbol{\theta}_0$ and iteratively updates the estimate by solving a *linearized* least squares problem. The method can be derived by considering a general objective function of the form:

$$f(\boldsymbol{\theta}) = \sum_{i}^{m} r_i^2(\boldsymbol{\theta}) \tag{2}$$

Here, r_i are called the *residuals* and the goal is to find $\boldsymbol{\theta}$ that minimizes f. This can be approximated by a linear least squares problem by linearizing r_i in the vicinity of $\boldsymbol{\theta}$ for a small step $\boldsymbol{\delta}$:

$$f(\boldsymbol{\theta} + \boldsymbol{\delta}) \approx g(\boldsymbol{\delta}) = \sum_{i}^{m} (r_i(\boldsymbol{\theta}) + \nabla r_i(\boldsymbol{\theta})\boldsymbol{\delta})^2 \tag{3}$$

where ∇r_i is the vector of first partial derivatives of r_i (also called the *gradient*). The new objective function g has the form of a linear least squares objective function in terms of $\boldsymbol{\delta}$ when $\boldsymbol{\theta}$ is fixed. It's shown in the curriculum that the solution $\boldsymbol{\delta}$ that minimizes g is the solution to the following linear system of equations:

$$\mathbf{J}^T\mathbf{J}\boldsymbol{\delta} = -\mathbf{J}^T\boldsymbol{r} \tag{4}$$

where

$$\mathbf{J} = \begin{bmatrix} \frac{\partial r_1}{\partial \theta_1} & \cdots & \frac{\partial r_1}{\partial \theta_n} \\ \vdots & & \vdots \\ \frac{\partial r_m}{\partial \theta_1} & \cdots & \frac{\partial r_m}{\partial \theta_n} \end{bmatrix} = \begin{bmatrix} \nabla r_1(\boldsymbol{\theta}) \\ \vdots \\ \nabla r_m(\boldsymbol{\theta}) \end{bmatrix} \tag{5}$$

is the *Jacobian matrix* consisting of the partial derivatives of the residuals, and

$$\boldsymbol{r}(\boldsymbol{\theta}) = (r_1(\boldsymbol{\theta}), \cdots, r_N(\boldsymbol{\theta}))^T \tag{6}$$

is the *residual vector*.

The system of equations (4) are called the *normal equations* and can be solved for $\boldsymbol{\delta}$ using standard linear algebra techniques. In Python/Numpy one can use `numpy.linalg.solve`. In Matlab, one can use the backslash operator. The Gauss-Newton method then updates the current estimate by moving some amount in the direction of $\boldsymbol{\delta}$:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha\boldsymbol{\delta} \tag{7}$$

where α is the *step size*. The process is then repeated at the new estimate: forming a new linearized least squares problem and solving for another step.

Note that using $\alpha = 1$ is optimal only if the linearization is exact. Depending on the objective function, the accuracy of the linearization can be quite poor, and a smaller step size should be used. A proper Gauss-Newton implementation will perform a so-called line search to determine the best step size, but we'll see another way to handle this later in the exercise (Levenberg-Marquardt).

1 Gauss-Newton

The following dataset is included in the zip:

- `data/video0000.jpg...video0360.jpg`: Image sequence (undistorted).
- `data/cameraK`: Camera intrinsic matrix $\mathbf{K}$.
- `data/model.txt`: Homogeneous marker coordinates in the helicopter model.
- `data/pose.txt`: Transformation $\mathbf{T}_{\text{platform}}^{\text{camera}}$ from base platform to the camera.
- `data/markers.txt`: Detected markers. The j'th row corresponds to the j'th image, and contains 7 tuples of the form (m_i, u_i, v_i), where m_i is 1 if marker i was detected or 0 otherwise, and (u_i, v_i) is the marker's pixel coordinates.
- `data/logs.txt`: Recorded angles from encoders. Each row contains a timestamp (in seconds), yaw, pitch and roll. The logs have been synchronized with the video.

The included template code takes care of reading the dataset for you and generating the final figure. Your job is to finish the implementation of the stub functions.

(a) Write a function that computes the residual vector $\boldsymbol{r}$ given the yaw, pitch and roll. Assume that $\hat{\boldsymbol{u}}_i = (\hat{u}_i, \hat{v}_i)$ is given by the ideal pinhole projection

$$\hat{u}_i = c_x + f_x X_i^c / Z_i^c \tag{8}$$
$$\hat{v}_i = c_y + f_y Y_i^c / Z_i^c \tag{9}$$

where $\mathbf{X}_i^c = (X_i^c, Y_i^c, Z_i^c)$ is the i-th marker in the model transformed to camera coordinates using the rigid-body transformations from Exercise 1:

$$\mathbf{X}_i^c = \begin{cases} \mathbf{T}_{\text{arm}}^{\text{camera}}(\psi, \theta)\mathbf{X}_i^{\text{arm}} & i \in \{0, 1, 2\} \\ \mathbf{T}_{\text{rotors}}^{\text{camera}}(\psi, \theta, \phi)\mathbf{X}_i^{\text{rotors}} & i \in \{3, 4, 5, 6\} \end{cases} \tag{10}$$

Note that not every marker is detected in each image, and the corresponding $\boldsymbol{u}_i$ are invalid. To handle this you should multiply the residual vector by a vector of weights that are 1 (or 0) if the marker was detected (or not).

Test your function on the first image. Initialize with the already-optimal estimates $\psi = 11.6°, \theta = 28.9°, \phi = -0.6°$ and check that your residuals are close to zero.

(b) Write a function that computes $\mathbf{J}^T\mathbf{J}$ and $\mathbf{J}^T\boldsymbol{r}$ (the normal equation terms). You may use finite differences to approximate the gradients ∇r_i. See also note [2].

(c) Implement the Gauss-Newton method. Run the method **up to image 86**. Use a constant step size $\alpha = 0.25$ and choose an appropriate number of iterations. Plot your estimated ψ, θ, ϕ against the logged encoder values (the provided `main` script generates the required figure). **Include the plot in your submission.**

(d) Run the method on image 87 (`data/video0087.jpg`). You should get an error or a warning complaining about $\mathbf{J}^T\mathbf{J}$ being singular. Explain what is happening.

2 Levenberg-Marquardt

Levenberg-Marquardt is a small modification of the Gauss-Newton method that greatly improves its practical applicability. The only difference is setting the step size $\alpha = 1$ and replacing the normal equations (4) by:

$$(\mathbf{J}^T\mathbf{J} + \lambda\mathbf{D})\boldsymbol{\delta} = -\mathbf{J}^T\boldsymbol{r} \tag{11}$$

where $\mathbf{D}$ is a positive definite matrix (in this exercise you can assume it's the identity) and $\lambda > 0$ is a scalar determined by the following rules:

1. At the start of optimization, λ is initialized to a relatively small value. A commonly used heuristic is 10^{-3} times the average of the diagonal elements of $\mathbf{J}^T\mathbf{J}$.
2. In each iteration, if the value of $\boldsymbol{\delta}$ obtained by solving (11) leads to a reduced error, then the step is accepted and λ is decreased (e.g. divided by 10) before the next iteration.
3. Otherwise, if the value $\boldsymbol{\delta}$ leads to an increased error, then λ is increased (e.g. multiplied by 10) and the normal equations are solved again. This is repeated until a value of $\boldsymbol{\delta}$ is found that leads to a reduced error.

It's also a good idea to add a *termination condition* to prevent unecessary iterations. A common choice is to stop when the change in the estimates between two successive iterations is less than the desired precision: $||\boldsymbol{\theta}_{n+1} - \boldsymbol{\theta}_n|| < \texttt{xtol}$.

(a) Implement the Levenberg-Marquardt (LM) method. Initialize $\psi = \theta = \phi = 0$ and test it on the first image. Use a maximum of 100 iterations and print the estimates along with λ in each iteration. What happens with λ? How many iterations does it take before the estimates stabilize to 0.001 radians precision?

(b) Run the LM method on image 87. Do you still get an error? Why/Why not?

(c) Run the LM method on the entire image sequence and plot your estimates of ψ, θ, ϕ against the logged encoder values. **Include the plot in your submission.**

Optional task (Not required to get exercise approved)

(d) **Make a video of your results!** Plot the coordinate frames and predicted versus observed marker points as in Fig. 1 and save each figure as a new image sequence. To convert this into a video, we can use ffmpeg. Assuming you saved your images as `out0.jpg`, `out1.jpg`, ..., `out<n>.jpg`, you can use this command to create a video:

```
ffmpeg -r 16 -i out%d.jpg -vcodec libx264 -pix_fmt yuv420p out.mp4
```

Note that you have to run the command using the terminal (Linux and Mac) or the command line prompt (Windows), while in the same folder as the images.

Notes

[1] The following papers are not required reading in the course, but they are good references on visual object tracking, including marker-based methods (as in this exercise) and marker-less methods:

- *Monocular Model-Based 3D Tracking of Rigid Objects: A Survey* by Vincent Lepetit and Pascal Fua
- *Pose Estimation for Augmented Reality: A Hands-On Survey* by Eric Marchand, Hideaki Uchiyama, and Fabien Spindler.

[2] Partial derivatives can be computed by deriving analytical expressions by hand and transcribing the expressions to code. Symbolic processing software, like Matlab or SymPy in Python, can also automatically derive the analytical expression for you (see Appendix E in *Robotics, Vision and Control*), although these are usually not simplified that well. There is also the option of automatic differentiation, which is the ability of the language or a library to automatically compute the partial derivative of a function with respect to its inputs. However, the simplest way to compute derivatives may be the finite differences method. For a function of a single scalar variable, the two-point finite difference approximation is

$$\frac{\partial f(x)}{\partial x} \approx \frac{f(x+\epsilon) - f(x)}{\epsilon} \tag{12}$$

where ϵ is a small change in x. For vector-variable functions the same method is applied to each variable in turn. The challenges associated with finite differences is that it may be slower than an optimized analytical expression, and a poor choice of ϵ can lead to instability.