TTK4255: Robotic Vision

Midterm project: Model-based pose estimation

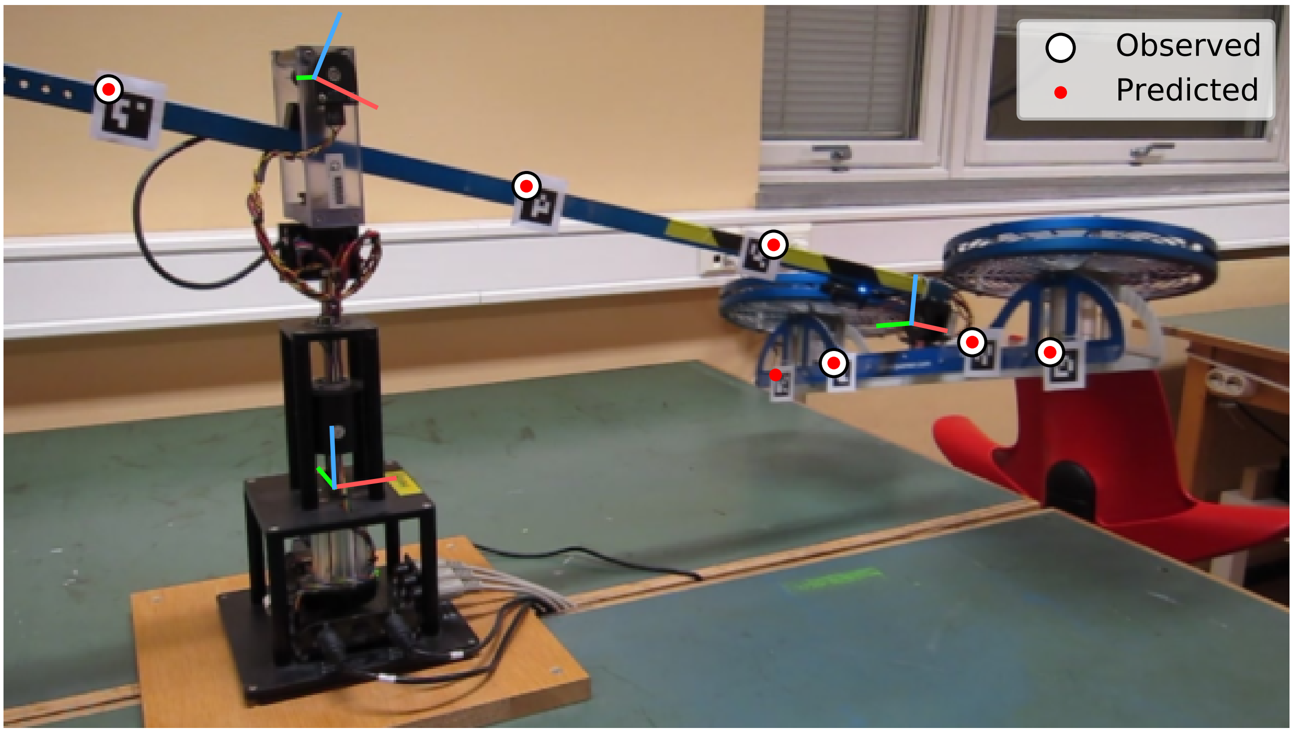

Figure 1: Results for a single image, showing the estimated coordinate frames, in addition to the observed and predicted location of the markers. The markers here are called AprilTags; these can be robustly detected thanks to their high contrast and error-tolerant coding system. Each marker is coded with a unique ID, which is used to establish 2D-3D correspondences.

Instructions

- The project can be done alone or in groups of two.
- Your project is approved/not approved. Your project is approved if you complete 60 percent of the tasks. The grading is explained in the “About” document on Blackboard under Course work.
- If you are stuck, you can often move on to the next task. If you don’t have a definitive answer, you may still get a partial score by describing your attempt so far and/or your thinking process.
- Upload a written report containing answers to the questions and requested program outputs within the deadline (see Blackboard). The report must be a single PDF. You must also upload a zip containing your code. For groups, only one of you should submit.
- You are not permitted to collaborate with other groups or copy solutions from previous years. Unlike the homework, we will look for plagiarism and non-permitted collaboration.

Getting help

We expect that you attempt to answer the tasks as they are formulated, without further guidance from the staff or fellow students (apart from your teammate). During the project period and up to the deadline, the Piazza forum will therefore only accept private posts. You may still post questions, but these should be primarily for reporting errors or requesting clarification about the requirements or grading. You are also welcome to ask questions about programming that you think are not specific to the course. Staff will still attend the digital and in-person help sessions.

About the project

You should be familiar with the Quanser 3-DOF Helicopter from Homework 3. As you will recall, the helicopter consists of an arm and a rotor carriage with three rotational degrees of freedom:

- Yaw (ψ): Rotation around an axis perpendicular to the mounting platform.
- Pitch (θ): Rotation of the arm up or down.
- Roll (ϕ): Rotation of the rotor carriage around the arm.

In this project you will estimate these angles in a pre-recorded image sequence of one of the helicopters in the lab at ITK. During recording, the helicopter was augmented with fiducial markers, which makes it easy to establish 2D-3D point correspondences. Instead of deriving a linear algorithm to estimate the angles, you will use the more versatile approach of iteratively minimizing the reprojection error between the markers’ predicted and detected image locations. To minimize the reprojection error you will use non-linear least squares optimization, which is frequently used in the robotic vision community. In Part 2 and Part 3 you will further investigate the use of optimization to estimate the pose of the camera and to calibrate the helicopter model itself.

Resources

Non-linear least squares is used in Szeliski 6 (Feature-based alignment) and 7 (Structure from motion), but these chapters can be hard to follow without a background in optimization theory. A summary of the necessary optimization theory is therefore given in Part 1. The booklet by Madsen, Nielsen and Tingleff (available on Blackboard) is otherwise a good introduction to non-linear least squares. A further overview of vision-related applications of non-linear least squares, along with some more advanced topics, can be found in Hartley and Zisserman Appendix 6 (available on Blackboard).

Provided data and code

You will do all the tasks on the dataset included in the zip, which is described below. Code is provided to you for loading the dataset in Python and Matlab, so you shouldn't need to read this too carefully.

File name(s)	Description
video<0000...0350>.jpg	Image sequence of helicopter undergoing motion. The images have had lens distortion removed, such that they satisfy a pinhole camera model with the provided intrinsic matrix $\mathbf{K}$
logs.txt	Recorded angles from the helicopter's encoders. Each row contains a timestamp (in seconds), followed by yaw, pitch and roll angles. The logs have been time-synchronized with the images.
detections.txt	Marker detections. The j'th row corresponds to the j'th image, and contains 7 tuples of the form (w_i, u_i, v_i), where $w_i = 1$ if marker i was detected and 0 otherwise, and (u_i, v_i) is the marker's pixel coordinates. (Note: w is a weight and not the homogeneous component $\tilde{w}$.)
K.txt	Camera intrinsic matrix $\mathbf{K}$.
heli_points.txt	Homogeneous 3D coordinates of markers in the helicopter model.
platform_to_camera.txt	Transformation matrix $\mathbf{T}^{\text{camera}}_{\text{platform}}$.
platform_corners_metric.txt	Metric coordinates of four points on the platform.
platform_corners_image.txt	Measured pixel coordinates of the above points.

Description of the included dataset.

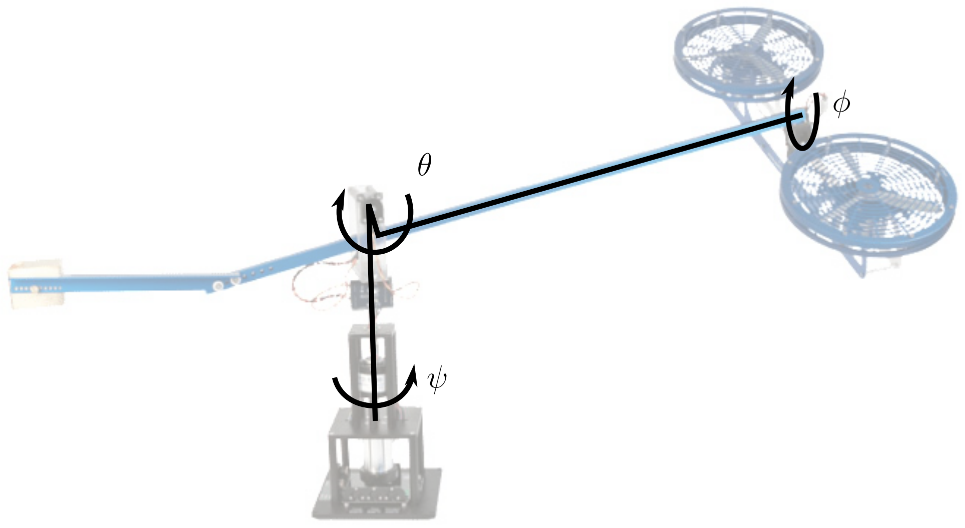

Figure 2: Quanser 3-DOF helicopter and its three degrees of freedom. The arrows indicate the direction of increasing yaw, pitch and roll.

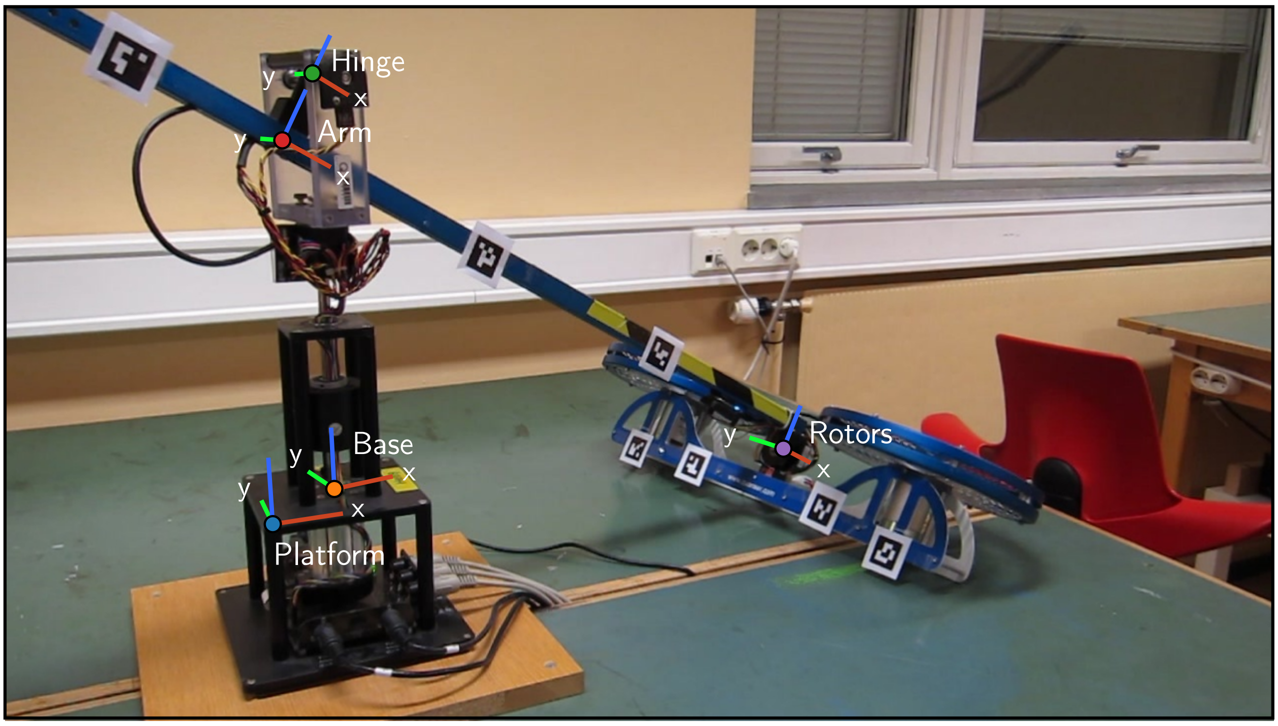

Figure 3: Helicopter coordinate frames

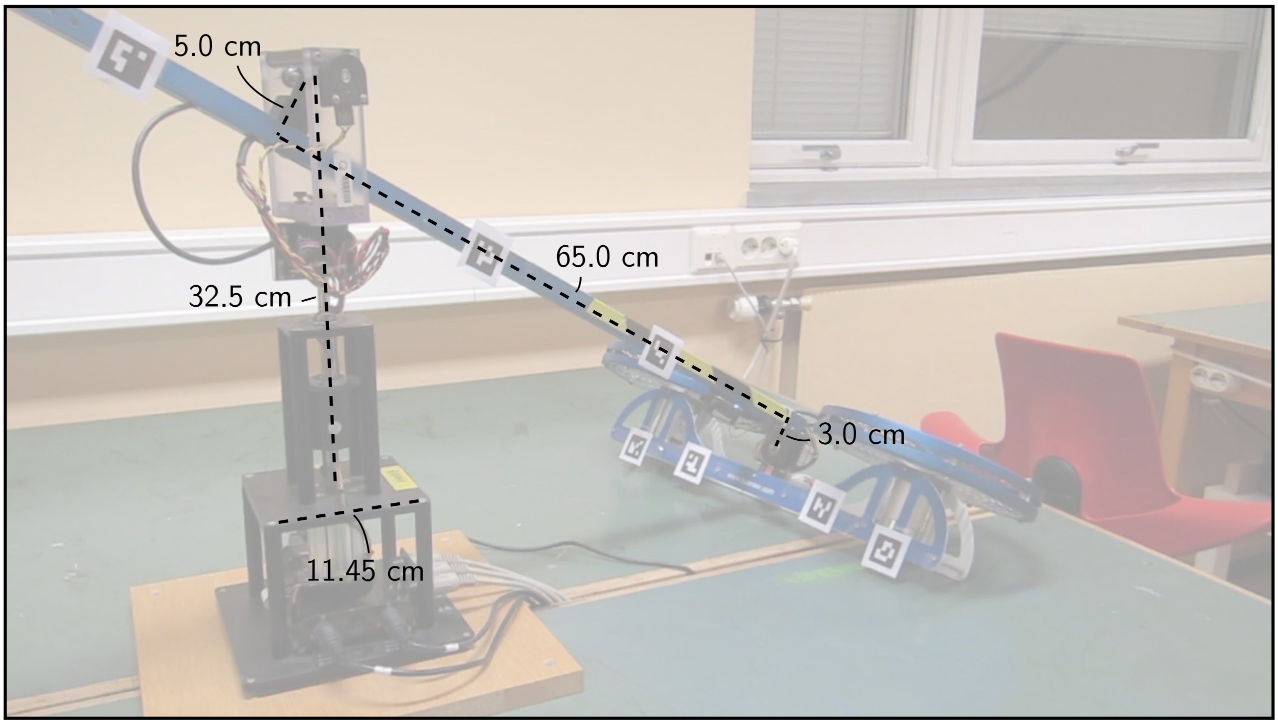

Figure 4: Helicopter dimensions

Part 1 Estimate the helicopter angles (45%)

In this part you will use an iterative algorithm to estimate the helicopter's angles. Compared with linear algorithms, this algorithm is more flexible and can easily incorporate constraints and other knowledge, such as a motion model or information from other sensors. The algorithm assumes that an optimal estimate of the angles is one that causes the predicted image locations of the markers to be close to their observed locations. This criterion is formulated as a least squares problem, where the quantity to be minimized is the sum of squared reprojection errors,

$$E(\mathbf{p}) = \sum_i ||\hat{\mathbf{u}}_i(\mathbf{p}) - \mathbf{u}_i||^2. \tag{1}$$

Here, E is the *cost* or *objective* function, $\mathbf{p}$ contains the unknown parameters that we want to estimate, and $\hat{\mathbf{u}}_i(\mathbf{p})$ and $\mathbf{u}_i$ are corresponding pairs of predicted and observed image locations. For the Quanser helicopter, the parameter vector is $\mathbf{p} = (\psi, \theta, \phi)$ and $\hat{\mathbf{u}}_i$ is given by the pinhole projection,

$$\tilde{\hat{\mathbf{u}}}_i = \mathbf{K}\mathbf{X}_i^c, \tag{2}$$

where $\mathbf{X}_i^c$ is the i'th marker's coordinates in the helicopter model transformed into the camera frame,

$$\mathbf{X}_i^c = \begin{cases} \mathbf{T}_{\text{arm}}^{\text{camera}}(\psi, \theta)\mathbf{X}_i^{\text{arm}} & i \in \{1, 2, 3\}, \\ \mathbf{T}_{\text{rotors}}^{\text{camera}}(\psi, \theta, \phi)\mathbf{X}_i^{\text{rotors}} & i \in \{4, 5, 6, 7\}. \end{cases} \tag{3}$$

Likewise, $\mathbf{u}_i$ is the observed image location of the i'th marker. Although the markers have a square shape and could therefore provide four correspondences each, you will only use a single corner of each marker. These points are shown in the front page figure.

Because $\hat{\mathbf{u}}_i(\mathbf{p})$ is a non-linear function of $\mathbf{p}$, this is called a non-linear least squares problem. Unlike linear least squares, there is no universally good algorithm to solve for the global minimum. Instead, the standard approach is to use an iterative optimization algorithm. Being "iterative" means to start from an initial estimate of the solution, and refine it over a number of iterations so as to decrease the objective function. Such methods typically only guarantee convergence to a local minimum.

The Matlab Optimization Toolbox and IPOPT are well-tested packages for solving generic non-linear optimization problems. Ceres, GTSAM and g2o are packages developed specifically for non-linear least squares problems that arise in computer vision, and utilize their special structure for efficiency. The basis for these latter packages are the Gauss-Newton and Levenberg-Marquardt methods, which you will work with here. The following pages therefore contain a brief review of these methods.

A simplified implementation of the Gauss-Newton method is included in the zip, and you should use an existing implementation for the Levenberg-Marquardt method. Python users may use `scipy.optimize.least_squares` (link) from Scipy (link). Matlab users may use `lsqnonlin` (link) from the Optimization toolbox (link). Scipy documentation in PDF form is included in the zip, as the documentation website is often down. In Matlab you can run "doc lsqnonlin" to see the documentation.

The Gauss-Newton method and basic least-squares terminology

The derivation of the Gauss-Newton method is enlightening for understanding the more widely applied method of Levenberg and Marquardt. First, it will be convenient to write our objective function as

$$E(\mathbf{p}) = \sum_{i=1}^{n} r_i(\mathbf{p})^2 \tag{4}$$

where $r_i(\mathbf{p})$ is called a *residual*, and is a scalar-valued function of the parameters. The key idea in both methods is to linearize each residual using its Taylor expansion around a running estimate $\hat{\mathbf{p}}$. At each iteration, this gives a local quadratic approximation of E of the form

$$E(\hat{\mathbf{p}} + \boldsymbol{\delta}) \approx \sum_{i=1}^{n} \left(r_i(\hat{\mathbf{p}}) + \frac{\partial r_i}{\partial \mathbf{p}}(\hat{\mathbf{p}})\boldsymbol{\delta} \right)^2 := \hat{E}(\boldsymbol{\delta}) \tag{5}$$

where $\boldsymbol{\delta}$ is a small step. The local approximation $\hat{E}(\boldsymbol{\delta})$ has the form of a *linear* least squares objective function, with $\boldsymbol{\delta}$ as the variables, and is therefore much easier to minimize. The Gauss-Newton step is the minimizer of $\hat{E}(\boldsymbol{\delta})$, which can be obtained by solving the linear system

$$\mathbf{J}^T\mathbf{J}\boldsymbol{\delta}^{\text{GN}} = -\mathbf{J}^T\mathbf{r} \tag{6}$$

where $\mathbf{r} = [r_1(\hat{\mathbf{p}}) \ \cdots \ r_n(\hat{\mathbf{p}})]^T$ is the vector containing all the residuals and $\mathbf{J}$ is the *Jacobian* containing their partial derivatives evaluated at $\hat{\mathbf{p}}$,

$$J_{ij} = \frac{\partial r_i}{\partial p_j}(\hat{\mathbf{p}}). \tag{7}$$

For example, if $\mathbf{r} \in \mathbb{R}^n$ and $\mathbf{p} \in \mathbb{R}^3$ then $\mathbf{J} \in \mathbb{R}^{n\times 3}$:

$$\mathbf{J} = \begin{bmatrix} \partial r_1/\partial p_1 & \partial r_1/\partial p_2 & \partial r_1/\partial p_3 \\ \vdots & \vdots & \vdots \\ \partial r_n/\partial p_1 & \partial r_n/\partial p_2 & \partial r_n/\partial p_3 \end{bmatrix}. \tag{8}$$

The matrix $\mathbf{J}^T\mathbf{J}$ is called the *approximate Hessian* and Eq. (6) are called the *normal equations*. These can be solved by standard linear algebra techniques, e.g. `numpy.linalg.solve` in Python or the backslash operator in Matlab, assuming that $\mathbf{J}^T\mathbf{J}$ is invertible. The Gauss-Newton method updates the current estimate by moving some amount α (the *step size*) in the direction of $\boldsymbol{\delta}^{\text{GN}}$

$$\hat{\mathbf{p}} \leftarrow \hat{\mathbf{p}} + \alpha\boldsymbol{\delta}^{\text{GN}} \tag{9}$$

and repeats the above process at the new estimate. If $\mathbf{J}^T\mathbf{J}$ is positive definite and $\mathbf{J}^T\mathbf{r} \neq \mathbf{0}$, then there exists a non-zero step size that decreases the objective function. However, a step size of 1 is optimal only if the linearization is exact. If the accuracy of the linearization is poor, this step may actually increase the objective function, even if $\mathbf{J}^T\mathbf{J}$ is positive definite, indicating that a smaller step size should have been used.

The Levenberg-Marquardt method

The Gauss-Newton method has some shortcomings. Mainly, it requires a strategy to determine a good step size, and there is the possibility that $\boldsymbol{\delta}$ is not a descent direction ($\mathbf{J}^T\mathbf{J}$ may not be positive definite), meaning that a good step size may not even exist. The Levenberg-Marquardt method is a small modification that addresses both issues simultaneously. The difference is that the step size is fixed to 1 and the normal equations are replaced with

$$(\mathbf{J}^T\mathbf{J} + \mu\mathbf{I})\boldsymbol{\delta}^{\text{LM}} = -\mathbf{J}^T\mathbf{r} \tag{10}$$

where $\mu > 0$ is a scalar called the *damping parameter*, which can both increase and decrease during the optimization. In a given iteration, if the step $\boldsymbol{\delta}^{\text{LM}}$ obtained by solving (10) leads to a reduced error,

$$E(\hat{\mathbf{p}} + \boldsymbol{\delta}^{\text{LM}}) < E(\hat{\mathbf{p}}), \tag{11}$$

then the step is accepted and μ is decreased before the next iteration. Otherwise, μ is increased and the normal equations are solved again. This is repeated until a step is found for the current iteration that leads to a reduced error. An interpretation of the damping parameter can be found in Hartley and Zisserman A6.2, p.601. Most important to note is that any positive value of μ guarantees that $\mathbf{J}^T\mathbf{J} + \mu\mathbf{I}$ is positive definite, and thereby that $\boldsymbol{\delta}^{\text{LM}}$ is a descent direction.

Termination conditions / stopping criterion

It's good practice to specify the *termination condition* or *stopping criterion*, to prevent too few or excessive iterations. Matlab has a page describing the possible stopping critera for its solvers (link), which are similar to those used in Scipy and other optimization packages. You will explore the use of one of these criteria in one of the tasks.

Computing partial derivatives

Partial derivatives can be computed by deriving analytical expressions by hand and transcribing the expressions to code. Symbolic processing software, like Matlab or SymPy, can automatically derive the analytical expression for you, 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, for this project, the most straightforward option may be the finite difference approximation, which is also used internally by `lsqnonlin` and `scipy.optimize.least_squares`. For a function of a scalar parameter, the "2-point" or "central" finite difference approximation of its derivative is

$$\frac{\partial f(p)}{\partial p} \approx \frac{f(p+\epsilon) - f(p-\epsilon)}{2\epsilon} \tag{12}$$

where ϵ is a small change in p. For functions of vector-valued parameters $\mathbf{p} \in \mathbb{R}^d$, the above formula is applied to each parameter $p_i, i = 1...d$, one at-a-time while keeping the other variables fixed. Note that setting ϵ either too high or too low can both lead to instability.

Task 1.1 (5%)

Ignore the rotor carriage and consider the helicopter as just the arm with two degrees of freedom (ψ, θ). Suppose only a single marker is observed and that this marker is on the arm. This gives a single 2D-3D correspondence. Argue that there then exists up to two physically realizable helicopter configurations (ψ, θ) that locally minimize the objective function. You may include a sketch to support your argument. A photograph of a paper drawing is fine.

Task 1.2 (5%)

In order to apply the described methods to our problem, we need to define the residuals $r_i(\mathbf{p})$. Given the objective function Eq. (1), you may be tempted to define the residuals as

$$\mathbf{r}(\mathbf{p}) = \Big[||\hat{\mathbf{u}}_1(\mathbf{p}) - \mathbf{u}_1|| \quad \cdots \quad ||\hat{\mathbf{u}}_M(\mathbf{p}) - \mathbf{u}_M|| \Big]^T, \tag{13}$$

where $M = 7$ is the number of markers, which gives one scalar residual per point correspondence. However, a better choice is to define two residuals per correspondence, one per horizontal difference and one per vertical difference,

$$\mathbf{r}(\mathbf{p}) = \Big[(\hat{u}_1(\mathbf{p}) - u_1) \quad \cdots \quad (\hat{u}_M(\mathbf{p}) - u_M) \quad (\hat{v}_1(\mathbf{p}) - v_1) \quad \cdots \quad (\hat{v}_M(\mathbf{p}) - v_M) \Big]^T. \tag{14}$$

Mathematically, both choices result in the exact same objective function, but give rise to very different linearizations. Speculate on why the linearization produced by the first choice might be problematic.

Task 1.3 (10%)

The hand-out code provides a class `Quanser`, which contains the helicopter model from HW3 and a utility function to visualize the frames and points. The partially implemented method `residuals` should compute the residuals $\mathbf{r}$ as described above. The result should be a vector of length $2M$, where M is the number of markers; the first M elements should be the horizontal differences and the last M elements should be the vertical differences.

Finish the implementation of `residuals` and run the `part1a` script without modification. It should print the residuals on image 0 using its optimal angles. Check that they are small (within $\pm$ 10 pixels) and include these numbers in your report. Next, modify `part1a` to estimate the angles for image 40 using Gauss-Newton with a step size of 0.9 and 10 steps. (You should only need to comment out two lines.) The script should generate a figure showing the reprojected frames and points, as well as the reprojection errors. Include this figure and the reprojection errors in your report.

The markers may not all be detected in every image. However, instead of letting the length of $\mathbf{r}$ change from image to image, it may be useful later to keep its length the same for each image, and handle invalid residuals by multiplying the corresponding entries of $\mathbf{r}$ by 0. The hand-out code provides a vector called `weights` that you can use to achieve this. The hand-out code has several tips specific to Python or Matlab that you are encouraged to read through.

Task 1.4 (5%)

Instead of using a fixed number of steps, modify the Gauss-Newton method to stop when the change in the parameters between two successive steps is small. For example, if $\mathbf{p}_{k-1}$ and $\mathbf{p}_k$ are two successive parameter vectors, then you stop when the Euclidean norm (or some other norm) of the difference, $||\mathbf{p}_k - \mathbf{p}_{k-1}||$, is less than a tolerance. This tolerance is often named `xtol` or `StepTolerance`. Using the same step size as before and a step tolerance of 0.01 degrees, how many steps does it take before the tolerance is reached? Does this change significantly for different initial values for $\mathbf{p}$?

Task 1.5 (5%)

In image 87 none of the markers on the rotor carriage are observed. If you run Gauss-Newton on this image, you should see a warning indicating that $\mathbf{J}^T\mathbf{J}$ is singular. Explain why.

Task 1.6 (5%)

A related issue is if the optimization problem is *over-parameterized*, whereby two or more parameters produce exactly, or nearly, the same change in $\mathbf{r}$ (up-to-scale). This may happen everywhere, or only at specific points in the parameter space. An example of an everywhere over-parameterized problem is if we were to introduce a fourth parameter, ψ_0, added to ψ in Eq. (3). An example of the latter, although not physically achievable, is if the helicopter were to point straight up, such that motion in ψ becomes nearly indistinguishable from ϕ. Describe what $\mathbf{J}$ and $\mathbf{J}^T\mathbf{J}$ would look like in these two examples, and the implications for the involved parameters.

Task 1.7 (5%)

The `part1b` script is set up to estimate the angles on the entire sequence using Levenberg-Marquardt. The script will generate a figure showing your estimated angles against the encoder logs, as well as the reprojection errors statistics. Modify the script to run correctly, and include the generated figures and the reprojection errors. Comment on the following in your report:

(a) What is the largest error between the encoder logs and the vision estimate?

(b) When is the error largest and when is the error smallest?

(c) When is the accuracy of the vision estimate of the different parameters best/worst?

(Tip: Read the comment in `plot_all.m/py` regarding initial offset correction.)

Task 1.8 (5%)

When using Levenberg-Marquardt from Scipy or Matlab, you will not get the warning from Task 1.5. Explain why and comment on the following in your report:

(a) In general, what does the Levenberg-Marquardt method do with parameters that do not have any observational data that would allow their estimation?

(b) In your opinion, is this a good way to handle the issue?

Part 2 Estimate the platform pose (20%)

In Part 1, the transformation $\mathbf{T}^{\text{camera}}_{\text{platform}}$ was given to you. Here you will explore how it can be estimated. As input, you receive the metric coordinates of four coplanar points on the platform (the same that you defined in Homework 3) and their corresponding pixel coordinates in the image. These are provided in matching order in `platform_corners_metric.txt` and `platform_corners_image.txt`. You do not need the helicopter model from Part 1 in this part. Only the four point correspondences and the calibration matrix are involved. The `part2` script also contains some tips and helper code.

Task 2.1 (5%)

Estimate $\mathbf{T}^{\text{camera}}_{\text{platform}}$ using the linear algorithm from Homework 4. Compute the predicted image location of the platform points using the following two transformations

$$\text{(a)}\ \tilde{\mathbf{u}} = \mathbf{K}\mathbf{H}[X\ Y\ 1]^T \qquad \text{(b)}\ \tilde{\mathbf{u}} = \mathbf{K}[\mathbf{R}\ \mathbf{t}][X\ Y\ 0\ 1]^T$$

where (X, Y) are the metric coordinates, $\mathbf{H}$ is the matrix returned by the direct linear transform, and $[\mathbf{R}\ \mathbf{t}]$ is the pose extracted from $\mathbf{H}$, with $\mathbf{R}$ being a valid rotation matrix. Include a figure of each set of predicted image locations, drawn on top of one of the images in the sequence. Also include their reprojection errors and explain why the reprojection errors in (a) should all be exactly zero, and why the reprojection errors in (b) are all, except for one point, non-zero.

Task 2.2 (10%)

Estimate $\mathbf{T}^{\text{camera}}_{\text{platform}}$ by minimizing the sum of squared reprojection errors with respect to $\mathbf{R}$ and $\mathbf{t}$, using Levenberg-Marquardt. Include a figure like the previous task and compare the reprojection errors. Tip: You will need to parameterize the pose as a vector. However, it is not obvious how to handle the rotation matrix due to the constraints between the entries (orthogonal and unit-length columns). Some options are described in Szeliski 2.1.4, but a simple solution is to use a *local parameterization* around a reference rotation matrix $\mathbf{R}_0$, e.g.

$$\mathbf{R}(\mathbf{p}) = \mathbf{R}_X(p_1)\mathbf{R}_Y(p_2)\mathbf{R}_Z(p_3)\mathbf{R}_0 \tag{15}$$

where p_1, p_2, p_3 are small angles. This parameterization suffers from gimbal lock in general, but for small angles it is fine. The chosen reference rotation $\mathbf{R}_0$ should therefore be close to the expected solution, for which the linear estimate may be a good guess. With three additional parameters for $\mathbf{t}$, you obtain a minimal (six-dimensional) parameterization of $\mathbf{T}^{\text{camera}}_{\text{platform}}$.

Task 2.3 (5%)

Suppose you only have three point correspondences instead of four. Show that there can then be more than one physically plausible transformation $\mathbf{T}^{\text{camera}}_{\text{platform}}$ that minimizes the reprojection error, by finding two different local minima. Include the 4×4 matrices in your report, along with a short description of how you found them and a figure of the axes projected into the image for each. Ensure that the transformations are both physically plausible (i.e. that the transformed points are in front of the camera).

Part 3 Calibrate the model via batch optimization (35%)

The helicopter model will not perfectly match reality. Besides inaccurate measurements of the lengths in Fig. 4 and the 3D marker locations, the structural assumptions may also be violated; the shaft may not be exactly perpendicular to the platform, the hinge may not be located exactly on the yaw axis, and so on. The consequence is that the estimate of the roll, pitch and yaw angles may be inaccurate, as well as limited in the achievable reprojection error (and followingly precision), despite perfect detections.

This can be addressed by performing *batch optimization* over multiple images, whereby these presumed constant aspects of the model are jointly estimated with the time-varying aspects. This is much like camera calibration, where the intrinsics (e.g. focal length) are estimated from multiple images under the assumption that the intrinsics do not change from image to image, while the extrinsics (the camera pose) can and should change. Likewise, the helicopter model has a set of *kinematic parameters*, that are constant over the entire image sequence, and a set of *state parameters* (ψ, θ, ϕ), that vary from image to image. The kinematic parameters describe the relative locations of the markers and how they are transformed into camera coordinates as a function of the state parameters. If the model is not over-parameterized, and if the helicopter undergoes sufficient motion, then there should be a unique parameter vector that minimizes the sum of reprojection errors over all the images,

$$E(\mathbf{p}) = \sum_{i=1}^{N} \sum_{m=1}^{M} w_{im} ||\hat{\mathbf{u}}_{im}(\mathbf{p}) - \mathbf{u}_{im}||^2, \tag{16}$$

where N and M is the number of images and markers, and w_{im}, $\hat{\mathbf{u}}_{im}$ and $\mathbf{u}_{im}$ is the weight, predicted location and detected location, for marker m in image i, and $\mathbf{p}$ now contains the kinematic parameters as well as the state parameters for every image, e.g. $\mathbf{p} = (\mathbf{p}_{\text{kinematic}}^T, \psi_1, \theta_1, \phi_1, ..., \psi_N, \theta_N, \phi_N)^T$.

In this part you will consider three models, A, B and C, that are more or less suited for calibration. In common is that they aim to estimate the 3D marker coordinates themselves (in the coordinate frame in which they are constant). Model A also aims to estimate the five lengths in Fig. 4. Model B and C are more general and can estimate small misalignments, such as the shaft not being exactly perpendicular to the platform. They achieve this by replacing the explicitly named frames in Fig. 3 with more or less freely oriented and positioned frames. (Note: Frames "2" and "3" play the same roles as "arm" and "rotors", respectively, but are not necessarily located near the same places.)

Model A (26 kinematic parameters): 3D marker locations and lengths in Fig. 4.

$$\begin{aligned}
\mathbf{T}_{\text{base}}^{\text{platform}} &= \mathbf{T}_{XYZ}(l_1, l_1, 0)\mathbf{R}_Z(\psi) \\
\mathbf{T}_{\text{hinge}}^{\text{base}} &= \mathbf{T}_{XYZ}(0, 0, l_2)\mathbf{R}_Y(\theta) \\
\mathbf{T}_{\text{arm}}^{\text{hinge}} &= \mathbf{T}_{XYZ}(0, 0, l_3) \\
\mathbf{T}_{\text{rotors}}^{\text{arm}} &= \mathbf{T}_{XYZ}(l_4, 0, l_5)\mathbf{R}_X(\phi)
\end{aligned}$$

Note: A "length" should be a positive number, but it would be unnecessarily complicated to introduce sign constraints for this project, so we will allow l_i to be negative.

Model B (33 kinematic parameters): 3D marker locations and 4D parameterizations of the frames.

$$\begin{aligned}
\mathbf{T}_1^{\text{platform}} &= \mathbf{R}_X(a_{X1})\mathbf{R}_Y(a_{Y1})\mathbf{T}_{XYZ}(l_{X1}, l_{Y1}, 0)\mathbf{R}_Z(\psi) \\
\mathbf{T}_2^1 &= \mathbf{R}_X(a_{X2})\mathbf{R}_Z(a_{Z2})\mathbf{T}_{XYZ}(l_{X2}, 0, l_{Z2})\mathbf{R}_Y(\theta) \\
\mathbf{T}_3^2 &= \mathbf{R}_Y(a_{Y3})\mathbf{R}_Z(a_{Z3})\mathbf{T}_{XYZ}(0, l_{Y3}, l_{Z3})\mathbf{R}_X(\phi)
\end{aligned}$$

Model C (39 kinematic parameters): 3D marker locations and 6D parameterizations of the frames.

$$\begin{aligned}
\mathbf{T}_1^{\text{platform}} &= \mathbf{R}_X(a_{X1})\mathbf{R}_Y(a_{Y1})\mathbf{R}_Z(a_{Z1})\mathbf{T}_{XYZ}(l_{X1}, l_{Y1}, l_{Z1})\mathbf{R}_Z(\psi) \\
\mathbf{T}_2^1 &= \mathbf{R}_X(a_{X2})\mathbf{R}_Y(a_{Y2})\mathbf{R}_Z(a_{Z2})\mathbf{T}_{XYZ}(l_{X2}, l_{Y2}, l_{Z2})\mathbf{R}_Y(\theta) \\
\mathbf{T}_3^2 &= \mathbf{R}_X(a_{X3})\mathbf{R}_Y(a_{Y3})\mathbf{R}_Z(a_{Z3})\mathbf{T}_{XYZ}(l_{X3}, l_{Y3}, l_{Z3})\mathbf{R}_X(\phi)
\end{aligned}$$

In Model B and C, the angles a_{Xi}, a_{Yi}, a_{Zi} and the displacements l_{Xi}, l_{Yi}, l_{Zi} are part of the kinematic parameters. The angles are expected to be close to zero.

Task 3.1 (10%)

Note: You can complete the other tasks in this part entirely with pen and paper, but it can be helpful to experiment with a working implementation to check your answers. Implement and perform batch optimization for each model. Modify `part1b` to use your calibrated models and include the generated figures and reprojection errors in your report. See the last page for more guidance.

Task 3.2 (5%)

Iterative optimization algorithms need an initial guess for the solution. For each model, suggest how you may initialize the kinematic and state parameters. (Tip: Be aware that $l_{X3} = 0$ in Model B, which you may also want to discuss in Task 3.5.)

Task 3.3 (5%)

For Model A, it turns out that there is not a unique solution (i.e. parameter vector that minimizes the objective function). However, a subset of the parameters may still have a unique solution. Is this true for the five lengths $l_{1...5}$?

Task 3.4 (5%)

For each model A–C, do any of the state parameters have a unique solution?

Task 3.5 (10%)

Is the smallest achievable value of the objective function the same for both Model B and Model C? In other words, are the additional parameters in Model C redundant? Support your answer with an explanation, for example using a simplified problem.

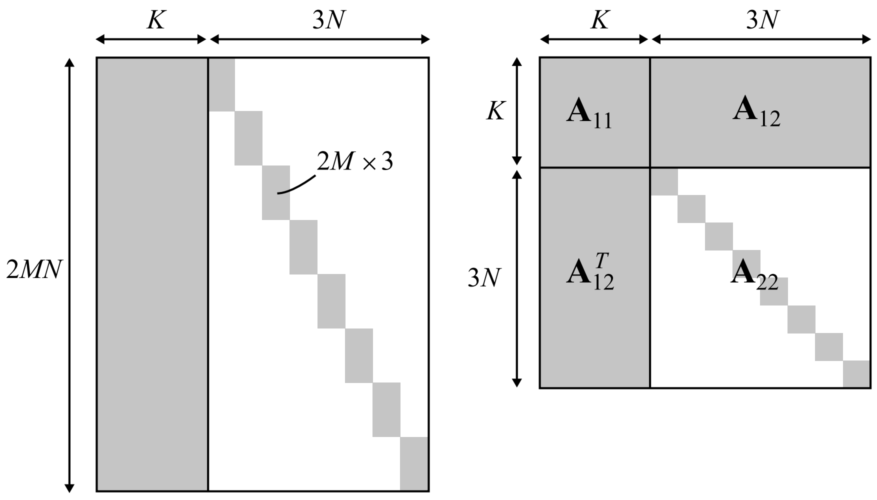

Figure 5: The "sparsity pattern" of the Jacobian $\mathbf{J}$ and the approximate Hessian $\mathbf{A} = \mathbf{J}^T\mathbf{J}$ for $N = 8$ images and K kinematic parameters. The shaded rectangles indicate blocks of possibly non-zero entries, while the unshaded areas are all zeros.

Tips for Task 3.1

Suppose that the helicopter model has K kinematic parameters. The angles can change in each image, so for N images there are $3N$ state parameters, giving a total of $K + 3N$ optimization variables. There are also $2M = 14$ scalar residuals per image, giving a total of $2MN$ scalar residuals. This is a much larger optimization problem than before, and while you could simply extend your solution from Part 1, it will be prohibitively slow. However, the problem turns out to have a similar structure as the *bundle adjustment* problem (Szeliski 7.4), and can be solved much faster by exploiting sparsity. In particular, notice that state variables from different images are independent; adjusting the helicopter angles for one image does not affect the reprojection error in a different image. This results in a sparse Jacobian (Fig. 5), which can be computed much faster by skipping elements known to be zero.

Both `lsqnonlin` and `scipy.optimize.least_squares` allow you to specify the Jacobian *sparsity pattern*, as a dense matrix of ones and zeros, which will speed up internal computations. However, Scipy (and possibly Matlab, depending on your version) does not allow you to specify the sparsity pattern while requesting the Levenberg-Marquardt algorithm. You should therefore drop the algorithm argument to request the default algorithm instead.

Note: The sparsity pattern must match how you define your residuals, specifically how you order the horizontal and vertical residuals for a single image, and how you order the residuals from all the images. Figure 5 may not match your ordering.