TTK4255: Robotic Vision

Homework 6: Optimal methods

Originally created by Simen Haugo, and modified by Mau Hing Yip.

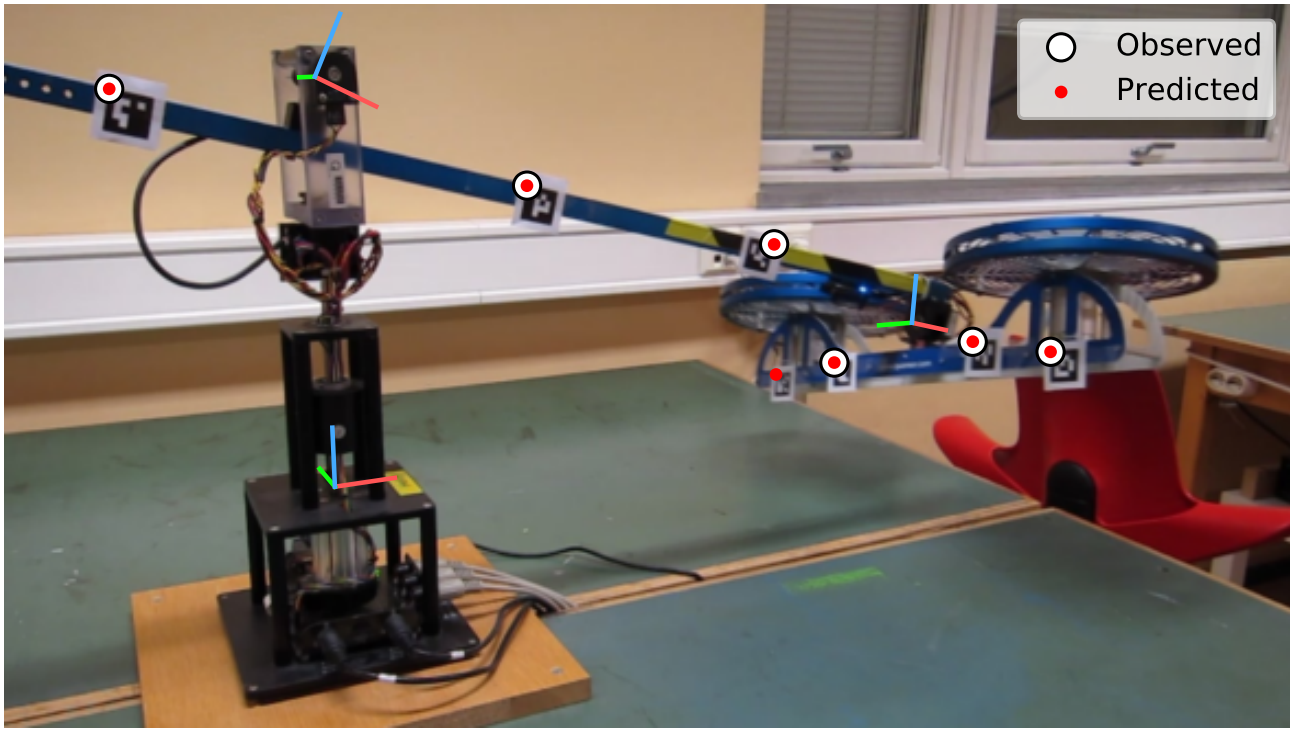

Figure 1: Output on a single image, showing the estimated coordinate frames, in addition to the observed and predicted marker locations. The markers here are called AprilTags; these can be robustly detected thanks to their high contrast and error-tolerant coding system. Each marker is coded with a unique ID, which simplifies the problem of establishing point correspondences.

Instructions

First make sure to read `about.pdf` in the "Course work" page on Blackboard. To get your assignment approved, you need to complete any 60%. Upload the requested answers and figures as a single PDF. You may collaborate with other students and submit the same report, but you still need to upload individually on Blackboard. Please write your collaborators' names on your report's front page. If you want detailed feedback, please indicate so on the front page.

About the assignment

Recall that the Quanser 3-DOF helicopter from Homework 4 has 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.

The main goal of this assignment is to estimate these angles in a pre-recorded image sequence of one of the helicopters in the ITK helicopter lab. The helicopter was augmented with AprilTag markers, which makes it easier to establish point correspondences.

There is no unique definition of "optimality", but for this assignment the estimates should minimize the sum of squared reprojection errors between predicted and measured image locations. This can be justified as being an instance of maximum likelihood estimation, with i.i.d. (independently and identically distributed) Gaussian noise in the measured image locations.

The resulting optimization problem generally does not have a universally good algorithm to find the global optimum. Instead, it's common practice to solve (at least part of) the problem using an iterative non-linear least squares algorithm (a "solver"), like Gauss-Newton or Levenberg-Marquardt, which you will explore here.

Relevant reading

We provide a brief review of the Gauss-Newton and Levenberg-Marquardt algorithms, which should be enough to complete the assignment. If you want to learn more, we can recommend the following resources (see learning materials on Blackboard):

- Madsen, Nielsen, and Tingleff. Methods for non-linear least squares problems.
- Nocedal and Wright. Numerical Optimization, 2nd edition: §10.3.
- Hartley and Zisserman: A6.
- Szeliski: §A.3.

Provided data and code

You will do all the tasks on the data included in the zip, which is described below. Code is provided to you for loading the data 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 been "undistorted" so as to satisfy a perspective 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 $\mathbf{T}_{\text{platform}}^{\text{camera}}$.
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.

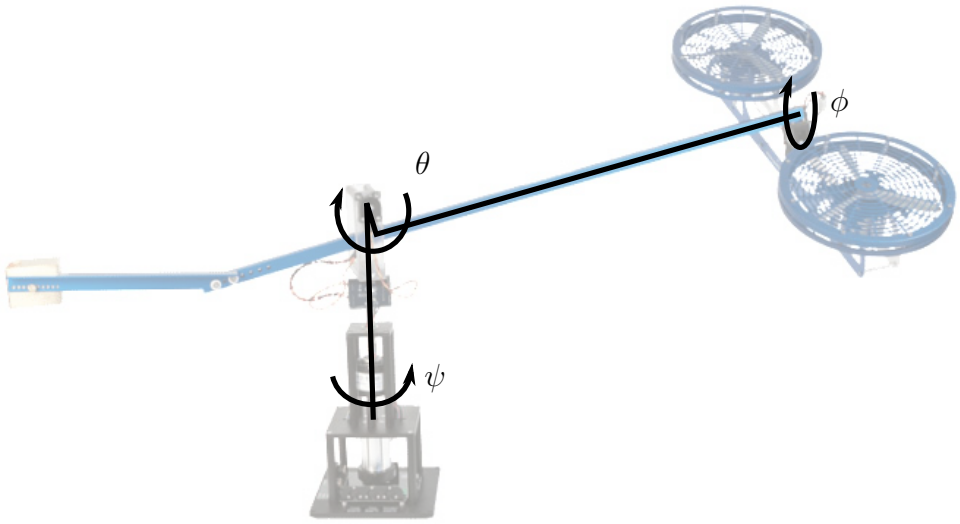

Figure 2: Quanser 3-DOF helicopter and its three degrees of freedom. The arrows indicate the direction of increasing yaw, pitch and roll.

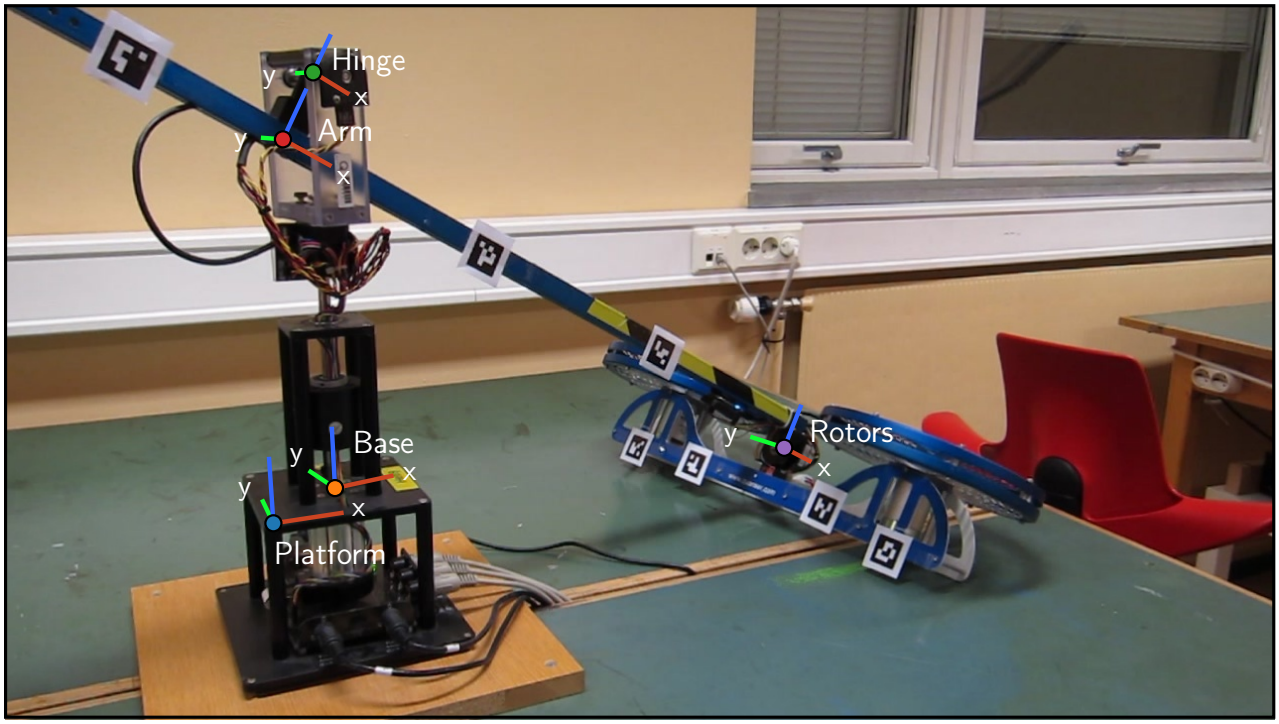

Figure 3: Helicopter coordinate frames

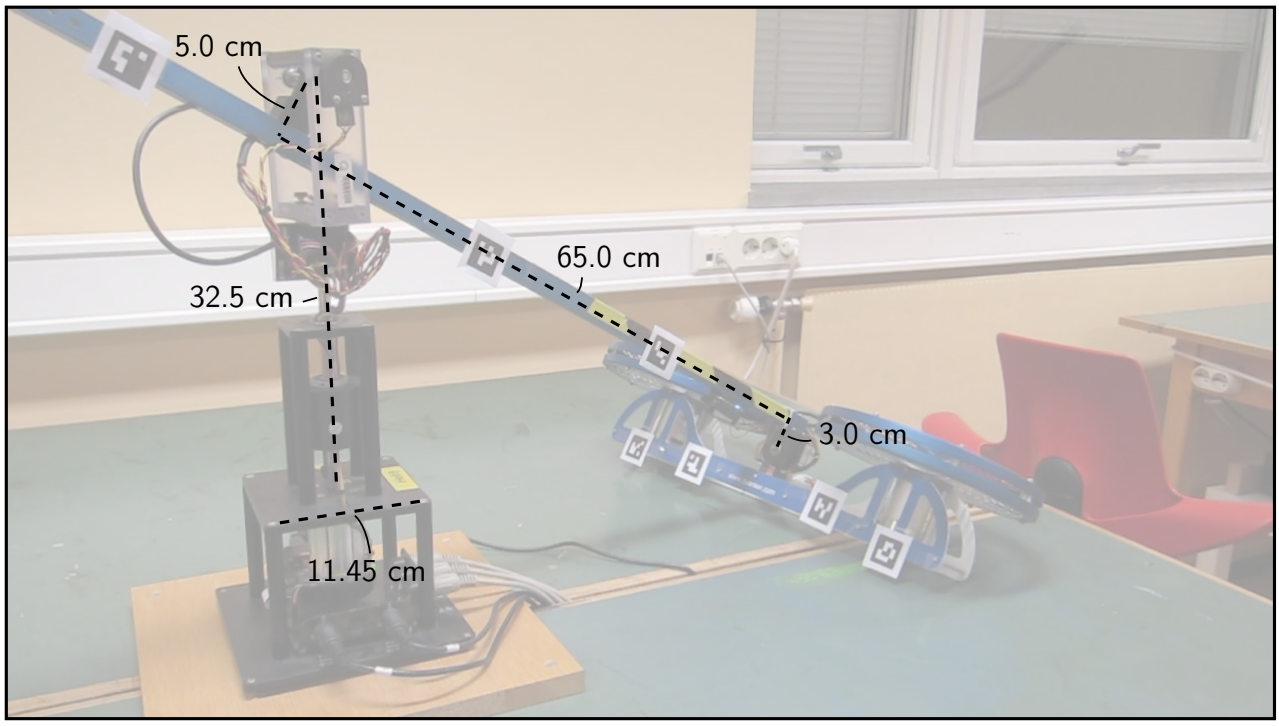

Figure 4: Helicopter dimensions

Brief review of Gauss-Newton (GN) and Levenberg-Marquardt (LM)

Implementations of Gauss-Newton and Levenberg-Marquardt expect to be given a function that takes a parameter vector $\mathbf{p}$, and computes a vector of scalars $\mathbf{r}(\mathbf{p}) = (r_1(\mathbf{p}), ..., r_n(\mathbf{p}))$, with the implicit assumption that the user wants to minimize the sum of squared entries of this vector,

$$E(\mathbf{p}) = ||\mathbf{r}(\mathbf{p})||_2^2 = \sum_{i=1}^{n} r_i(\mathbf{p})^2. \tag{1}$$

In optimization literature, the entries r_i are often called *residuals*. The key idea behind the Gauss-Newton algorithm is to linearize each residual (using the 1st-order partial derivatives) around the current estimate $\hat{\mathbf{p}}$, so as to obtain a *linear* least squares problem that can be solved easily. This is done iteratively, each time producing a small update to the parameters.

At each iteration, the linearization of r_i gives a local quadratic approximation of E:

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

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. The Gauss-Newton step is the minimizer of $\hat{E}(\boldsymbol{\delta})$, which can be obtained by solving the linear system

$$\mathbf{J}^\mathsf{T}\mathbf{J}\boldsymbol{\delta}^{\text{GN}} = -\mathbf{J}^\mathsf{T}\mathbf{r}, \tag{3}$$

where $\mathbf{r} = \mathbf{r}(\hat{\mathbf{p}})$ is the vector of residuals at the current estimate $\hat{\mathbf{p}}$, and $\mathbf{J}$ is a matrix (called the *Jacobian*) containing their partial derivatives evaluated at $\hat{\mathbf{p}}$:

$$\mathbf{J} = \begin{bmatrix} J_{11} & J_{12} & \cdots \\ \vdots & \vdots & \vdots \\ J_{n1} & J_{n2} & \cdots \end{bmatrix} \quad \text{where} \quad J_{ij} = \left.\frac{\partial r_i(\mathbf{p})}{\partial p_j}\right|_{\hat{\mathbf{p}}}. \tag{4}$$

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

$$\mathbf{J} = \begin{bmatrix} \partial r_1/\partial p_1 & \partial r_1/\partial p_2 & \partial r_1/\partial p_3 \\ \vdots & \vdots & \vdots \\ \partial r_n/\partial p_1 & \partial r_n/\partial p_2 & \partial r_n/\partial p_3 \end{bmatrix}. \tag{5}$$

The matrix $\mathbf{J}^\mathsf{T}\mathbf{J}$ is called the *approximate Hessian*. Equations 3 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 $\mathbf{J}^\mathsf{T}\mathbf{J}$ is invertible. The Gauss-Newton method updates the current estimate by moving some amount α (the *step size*) in the direction of $\boldsymbol{\delta}^{\text{GN}}$

$$\hat{\mathbf{p}} \leftarrow \hat{\mathbf{p}} + \alpha\boldsymbol{\delta}^{\text{GN}} \tag{6}$$

and repeats the above at the new estimate. If the linearization is exact, then $\alpha = 1$ makes the algorithm converge in a single step.

Levenberg-Marquardt

The Gauss-Newton algorithm requires a strategy to determine the step size, and it is possible that $\boldsymbol{\delta}$ is not a descent direction ($\mathbf{J}^\mathsf{T}\mathbf{J}$ may not be positive definite), meaning that no step size will decrease E. The Levenberg-Marquardt algorithm addresses both issues simultaneously. The difference is that the step size is fixed to 1 and the normal equations are replaced with

$$(\mathbf{J}^\mathsf{T}\mathbf{J} + \mu\mathbf{I})\boldsymbol{\delta}^{\text{LM}} = -\mathbf{J}^\mathsf{T}\mathbf{r}, \tag{7}$$

where $\mu > 0$ is a *damping parameter*, which can increase and decrease during the optimization. In a given iteration, if the step $\boldsymbol{\delta}^{\text{LM}}$ obtained by solving (7) leads to a reduced error, i.e.

$$E(\hat{\mathbf{p}} + \boldsymbol{\delta}^{\text{LM}}) < E(\hat{\mathbf{p}}), \tag{8}$$

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 $\mathbf{J}^\mathsf{T}\mathbf{J} + \mu\mathbf{I}$ is positive definite, and thereby that $\boldsymbol{\delta}^{\text{LM}}$ is a descent direction.

Termination conditions

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 assignment, 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

$$\left.\frac{\partial f(p)}{\partial p}\right|_p \approx \frac{f(p+\epsilon) - f(p-\epsilon)}{2\epsilon} \tag{9}$$

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 $p_i, i = 1...d$ in turn, while keeping the other variables fixed. Note that setting ϵ either too high or too low can both lead to instability.

Part 1 Estimate the helicopter angles (60%)

We will assume 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 often formulated as a least squares problem, where the quantity to be minimized is the sum of squared reprojection errors

$$E(\mathbf{p}) = \sum_i ||\hat{\mathbf{u}}_i(\mathbf{p}) - \mathbf{u}_i||_2^2 = \sum_i (\hat{\mathbf{u}}_i(\mathbf{p}) - \mathbf{u}_i)^\mathsf{T}(\hat{\mathbf{u}}_i(\mathbf{p}) - \mathbf{u}_i). \tag{10}$$

The notation $||\cdot||_2$ means the L_2-norm of a vector which is defined as in the right-most expression. Here, E is called the *cost* or *objective* function, $\mathbf{p}$ contains the parameters to be estimated, and $\hat{\mathbf{u}}_i(\mathbf{p})$ and $\mathbf{u}_i$ are corresponding pairs of predicted and observed image locations. In this part, the parameter vector is $\mathbf{p} = (\psi, \theta, \phi)$ and $\hat{\mathbf{u}}_i(\mathbf{p})$ should be computed using a perspective camera model,

$$\tilde{\hat{\mathbf{u}}}_i(\mathbf{p}) = \mathbf{K} \begin{bmatrix} \mathbf{I} & \mathbf{0} \end{bmatrix} \tilde{\mathbf{X}}_i^{\text{camera}}(\mathbf{p}), \tag{11}$$

where $\mathbf{X}_i^{\text{camera}}(\mathbf{p})$ is the i'th marker's coordinates in the camera frame, i.e.

$$\tilde{\mathbf{X}}_i^{\text{camera}}(\mathbf{p}) = \begin{cases} \mathbf{T}_{\text{arm}}^{\text{camera}}(\psi, \theta)\tilde{\mathbf{X}}_i, & i \in \{1, 2, 3\}, \\ \mathbf{T}_{\text{rotors}}^{\text{camera}}(\psi, \theta, \phi)\tilde{\mathbf{X}}_i, & i \in \{4, 5, 6, 7\}. \end{cases} \tag{12}$$

Likewise, $\mathbf{u}_i$ is the observed image location of the i'th marker. Although the marker detector provides four points per marker, the data you are given here only provides a single point per 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, a common 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 algorithms 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 geometric computer vision problems, 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.

You are not asked to implement Gauss-Newton or Levenberg-Marquardt. A simplified implementation of the Gauss-Newton method is in the zip, and you should use an existing implementation of Levenberg-Marquardt. Python users may use `scipy.optimize.least_squares` [link] from Scipy. Matlab users may use `lsqnonlin` [link] from the Optimization toolbox [link]. The hand-out code shows how you can call these library functions, but you may still want to read the documentation.

Task 1.1: (10%) Ignore the rotor carriage and consider just the arm, with its two degrees of freedom (ψ, θ). Suppose only a single marker is observed and that this marker is on the arm, giving just a single point correspondence. Argue that there then exists up to two physically achievable helicopter configurations (ψ, θ) that locally minimize Eq. (10). Both configurations should be achievable without breaking the helicopter (e.g. without rotating the arm past the hinge).

Task 1.2: (10%) The non-linear least squares solvers we recommend for this assignment expect to be given a function that computes and returns a vector of residuals. Thus, you do not provide a function that computes Eq. (10). Instead, the solvers assume that the sum of squared elements of this vector is the objective function that you want to optimize. This is partly what distinguishes a *least squares* solver from solvers for generic optimization problems. Defining the vector of residuals, and writing the function to compute it, is therefore one of the things you need to do.

The elements of the vector of residuals do not need to be the same as the terms in the sum in Eq. (10). Consider two potential definitions:

$$\text{(A)} \quad \mathbf{r}(\mathbf{p}) = \begin{bmatrix} ||\hat{\mathbf{u}}_1(\mathbf{p}) - \mathbf{u}_1|| \\ \vdots \\ ||\hat{\mathbf{u}}_M(\mathbf{p}) - \mathbf{u}_M|| \end{bmatrix} \quad \text{and} \quad \text{(B)} \quad \mathbf{r}(\mathbf{p}) = \begin{bmatrix} \hat{u}_1(\mathbf{p}) - u_1 \\ \vdots \\ \hat{u}_M(\mathbf{p}) - u_M \\ \hat{v}_1(\mathbf{p}) - v_1 \\ \vdots \\ \hat{v}_M(\mathbf{p}) - v_M \end{bmatrix}, \tag{13}$$

where $M = 7$ is the number of markers. When the elements of $\mathbf{r}$ are squared and summed, both definitions result in the exact same objective function.

(a) Show that the entries in (A) are not differentiable when the reprojection error is 0.

(b) We rarely expect to bring all of the reprojection errors to exactly zero. Therefore, besides potential numerical issues caused by floating point arithmetic, the non-differentiability at zero error should not be a problem. Yet, definition (B) is better suited for use in a NLS solver, even in a machine performing exact real number arithmetic. Explain why.

Task 1.3: (10%) The hand-out code provides a `Quanser` class, 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 $\mathbf{r}$. 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 pre-determined optimal angles. Check that the residuals 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. The script should generate a figure showing the reprojected frames and points, as well as the reprojection errors $||\hat{\mathbf{u}}_i - \mathbf{u}_i||$. 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 code 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 can stop when the L_2-norm of $\mathbf{p}_k - \mathbf{p}_{k-1}$ 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 $\mathbf{J}^\mathsf{T}\mathbf{J}$ is singular. Explain why.

Task 1.6: (10%) A related issue can occur if a subset of the parameters are locally indistinguishable from each other in their effect on $\mathbf{r}$, either everywhere or at specific points in the parameter space. In other words, the solution is locally ambiguous.

An example of this, although not physically achievable, is if the helicopter were to point straight up, such that motion in ψ becomes nearly indistinguishable from ϕ. Another example is if we replace every occurrence of ψ in Eq. (12) with $\psi + \psi_0$, where ψ_0 is an additional free parameter. Describe what $\mathbf{J}$ and $\mathbf{J}^\mathsf{T}\mathbf{J}$ would look like in these two examples, and the implications for the involved parameters.

Task 1.7: (5%) The `part1b` script estimates the angles on the entire sequence using Levenberg-Marquardt (LM). Modify the script to run correctly, and include the generated figures and outputs.

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

Task 1.8: (5%) When using LM, you will not get the warning from Task 1.5. Describe what happens with ϕ in images like 87 when using LM.

Part 2 Estimate the platform pose (15%)

In Part 1, the transformation $\mathbf{T}_{\text{platform}}^{\text{camera}}$ was given to you. Here you will explore how it can be estimated. As input, you get the metric coordinates of four points on the platform (the same that you defined in Homework 4) and their corresponding pixel coordinates in one image (which are identical in all images for this data). These are provided in matching order in `platform_corners_metric.txt` and `platform_corners_image.txt`. The `part2` script contains some tips and helper code.

> You do not need the helicopter model in this part — only the four point correspondences and the intrinsic matrix are involved.

The recommended solvers expect that the residual-computing function takes a vector of scalars, and therefore don't natively support 3D poses as optimization variables. You therefore need to somehow parameterize the pose as a vector. You may be tempted to flatten the entries of the rotation matrix and translation vector into a vector. However, rotation matrices have constraints between the entries (orthogonal and unit-length columns), which will not necessarily be respected by the solver.

While you can find solvers that support hard constraints, these can be less reliable than solvers for unconstrained problems. We can keep the problem unconstrained with an appropriate parameterization of 3D rotations. Some options are described in Szeliski §2.1.4, but a simple one is a *local parameterization* around a reference rotation matrix $\mathbf{R}_0$, e.g.

$$\mathbf{R}(\mathbf{p}) = \mathbf{R}_X(p_1)\mathbf{R}_Y(p_2)\mathbf{R}_Z(p_3)\mathbf{R}_0, \tag{14}$$

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. The translation does not require special care, and can be parameterized as a 3D vector. The parameter vector therefore has a total of 6 parameters.

Task 2.1: (10%) Estimate $\mathbf{T}_{\text{platform}}^{\text{camera}}$ by minimizing the sum of squared reprojection errors using Levenberg-Marquardt. Include a figure of the reprojected platform frame, and the predicted image locations of the platform corners, drawn on top of one of the images in the sequence.

You will need a reference rotation $\mathbf{R}_0$ and an initial estimate of the translation. For now, you can define these by guessing approximate values (later in the course we'll look at initialization-free methods). The platform frame is approximately 0.8 meters in front of the camera.

Task 2.2: (5%) If you only have three point correspondences instead of four, then there can be more than one physically plausible transformation $\mathbf{T}_{\text{platform}}^{\text{camera}}$ that minimizes the reprojection error. Physically plausible meaning that the transformed points are in front of the camera.

Prove this experimentally by finding two different 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 both transformations are physically plausible.

Part 3 Calibrate the kinematic model (25%)

Our helicopter model will not perfectly match reality. Besides inaccurate measurements of the lengths in Fig. 4 and the 3D marker locations, the kinematic model itself may also be inaccurate; the shaft is not exactly perpendicular to the platform, the hinge is not exactly on the yaw axis, and so on. This in turn leads to inaccurate joint angle estimates.

This can be addressed by treating the kinematic model itself as something to be estimated. We can formalize this by considering our model to have a set of *kinematic parameters*, that presumably are fixed over the entire recorded image sequence (and hopefully over future images too), and a set of *state parameters* (ψ, θ, ϕ), that vary from image to image. To estimate these, we want to use all the data available. We therefore define an objective function that optimizes over multiple images simultaneously:

$$E(\mathbf{p}) = \sum_{i=1}^{N} \sum_{m=1}^{M} w_{im} ||\hat{\mathbf{u}}_{im}(\mathbf{p}) - \mathbf{u}_{im}||^2, \tag{15}$$

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. $\mathbf{p} = (\mathbf{p}_{\text{kinematic}}^{\mathsf{T}}, \psi_1, \theta_1, \phi_1, ..., \psi_N, \theta_N, \phi_N)$.

If the helicopter undergoes sufficiently rich motion, then we may be able to tease apart the time-fixed parameters from the time-varying parameters. The resulting kinematic parameters can (hopefully) be used to obtain more accurate angle estimates, both on the same sequence and on future images.

This requires a parameterized kinematic model. Two models are described below. One possibility is to simply reuse the same model structure from Part 1, but let the five lengths and the marker coordinates be free parameters. This is Model A. The more general Model B replaces the explicitly named frames in Fig. 3 with three freely oriented and positioned frames (corresponding to the three rigid parts of the helicopter).

Model A (**26 kinematic parameters**):

$$\begin{aligned}
\mathbf{T}_{\text{base}}^{\text{platform}} &= \mathbf{T}_{XYZ}(l_1, l_1, 0)\mathbf{R}_Z(\psi), \\
\mathbf{T}_{\text{hinge}}^{\text{base}} &= \mathbf{T}_{XYZ}(0, 0, l_2)\mathbf{R}_Y(\theta), \\
\mathbf{T}_{\text{arm}}^{\text{hinge}} &= \mathbf{T}_{XYZ}(0, 0, l_3), \\
\mathbf{T}_{\text{rotors}}^{\text{arm}} &= \mathbf{T}_{XYZ}(l_4, 0, l_5)\mathbf{R}_X(\phi), \\
\mathbf{p} &= (\mathbf{X}_1^{\mathsf{T}}, ..., \mathbf{X}_M^{\mathsf{T}}, l_1, ..., l_5, \psi_1, \theta_1, \phi_1, ..., \psi_N, \theta_N, \phi_N),
\end{aligned}$$

Note: Lengths should be positive, but it would be unnecessarily complicated to introduce sign constraints, so we allow l_i to be negative.

Model B (**39 kinematic parameters**):

$$\begin{aligned}
\mathbf{T}_1^{\text{platform}} &= \mathbf{R}_X(a_{X1})\mathbf{R}_Y(a_{Y1})\mathbf{R}_Z(a_{Z1})\mathbf{T}_{XYZ}(l_{X1}, l_{Y1}, l_{Z1})\mathbf{R}_Z(\psi), \\
\mathbf{T}_2^1 &= \mathbf{R}_X(a_{X2})\mathbf{R}_Y(a_{Y2})\mathbf{R}_Z(a_{Z2})\mathbf{T}_{XYZ}(l_{X2}, l_{Y2}, l_{Z2})\mathbf{R}_Y(\theta), \\
\mathbf{T}_3^2 &= \mathbf{R}_X(a_{X3})\mathbf{R}_Y(a_{Y3})\mathbf{R}_Z(a_{Z3})\mathbf{T}_{XYZ}(l_{X3}, l_{Y3}, l_{Z3})\mathbf{R}_X(\phi), \\
\mathbf{p} &= (\mathbf{X}_1^\mathsf{T}, ..., \mathbf{X}_M^\mathsf{T}, a_{X1}, ..., l_{Z3}, \psi_1, \theta_1, \phi_1, ..., \psi_N, \theta_N, \phi_N).
\end{aligned}$$

Note: Frames 2 and 3 play the same roles as "arm" and "rotors", respectively, but are not necessarily located near them. The ordering of the transformations here is in no way essential.

Task 3.1: (5%) To initialize the parameter vector, you can use the trajectory generated in Part 1 for the state parameters. For Model A, you can initialize the kinematic parameters using the lengths and marker coordinates measured in Part 1.

Suggest initial values for the kinematic parameters in Model B (angles a_*, displacements l_*, marker coordinates $\mathbf{X}_1, .., \mathbf{X}_7$) so that $\mathbf{X}_1^{\text{camera}}, ..., \mathbf{X}_7^{\text{camera}}$ are identical to those computed by Model A, for all (ψ, θ, ϕ).

Task 3.2: (15%) Implement and calibrate each model. Use the calibrated model to estimate the joint angles again, as in Part 1, over the entire provided image sequence. Include the resulting state trajectories and the reprojection error statistics in your report.

Including all the provided images in the calibration will likely be intolerably slow, unless you follow the tip described on the last page of this document. You can still get decent results with a small subset of images (e.g. 5–10), preferably depicting the helicopter in significantly different configurations (e.g. 10 immediately successive images is not good).

Task 3.3: (5%) Try to find out experimentally if the kinematic and state parameters, for both models, have unique solutions.

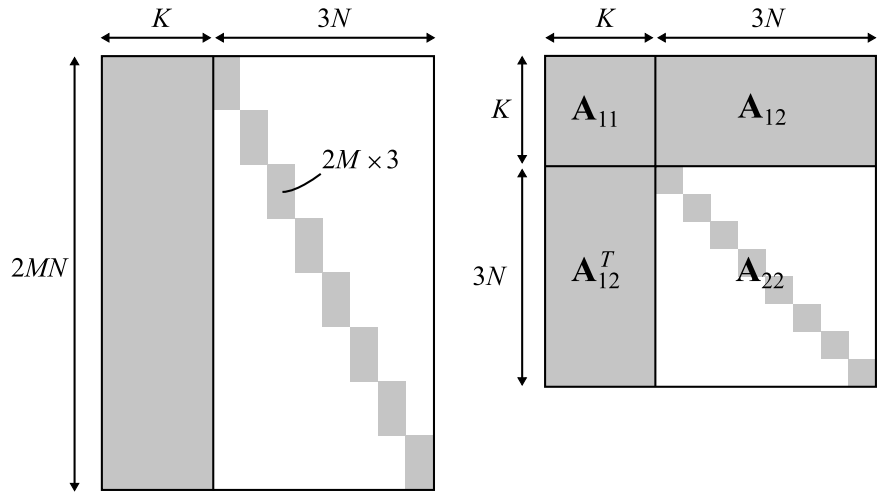

Figure 5: The "sparsity pattern" of the Jacobian $\mathbf{J}$ and the approximate Hessian $\mathbf{A} = \mathbf{J}^\mathsf{T}\mathbf{J}$ for $N = 8$ images and K kinematic parameters. The shaded rectangles indicate blocks of possibly non-zero entries, while the unshaded areas are all zeros.

Speeding things up

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 §11.4.2), 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. Note that 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.