TTK4255: Robotic Vision

Midterm project: Model-based pose estimation

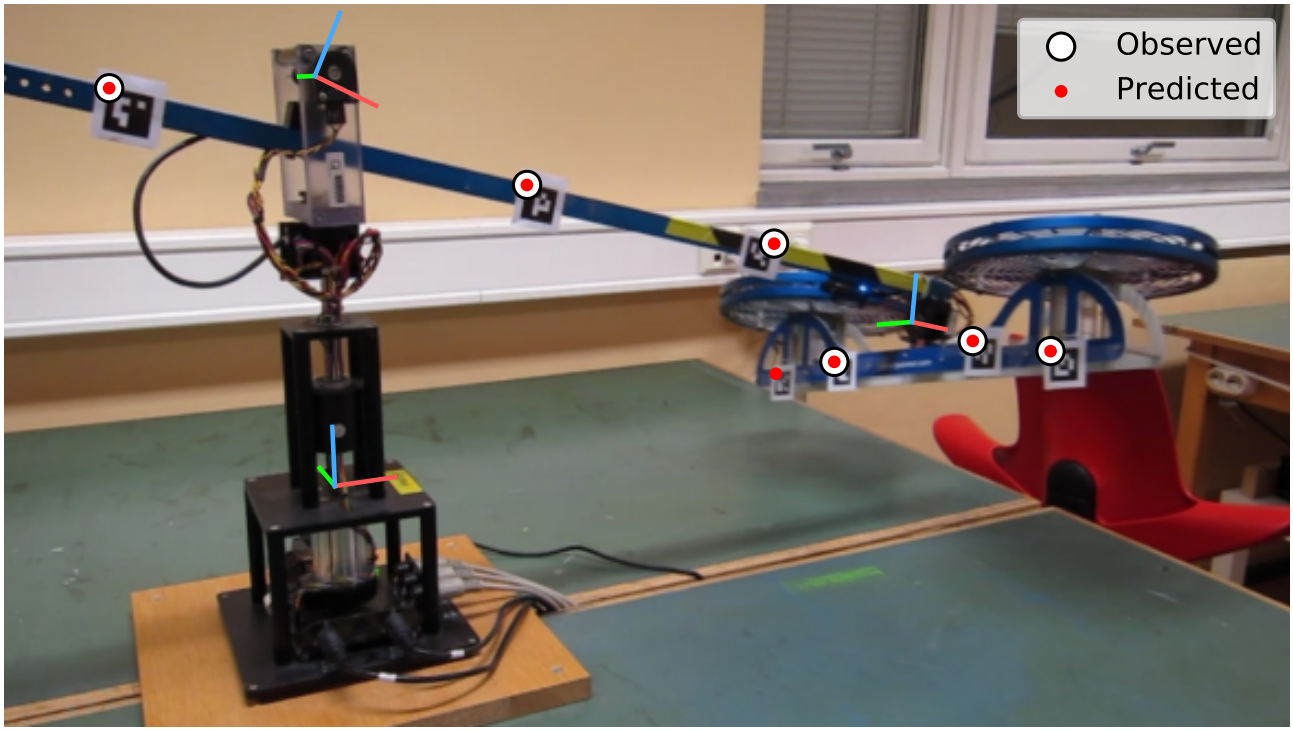

Figure 1: Results for a single image, showing the estimated coordinate frames, in addition to the observed and predicted location of the markers. The markers here are called AprilTags; these can be robustly detected thanks to their high contrast and error-tolerant coding system. Each marker is coded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

Instructions

- The project can be done alone or in groups of two (no more).
- Your score counts toward 25% of your final grade. In groups each member gets the same score.
- To receive a score, you must 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 member needs to upload everything.
- 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.**

> It is expected that you attempt to solve the problems 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 allow private posts. You may still post questions, but we may decide to not answer it, if it is too revealing. If we do answer a question, we will make it public. Questions should be primarily for reporting errors in the text or hand-out code, or requesting clarification about the requirements or grading.

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. However, instead of deriving a linear algorithm to estimate the angles, you will use the more versatile approach of directly minimizing the reprojection error between the markers' predicted and detected image locations.

To minimize the reprojection error you will implement the Gauss-Newton and Levenberg-Marquardt methods, which are frequently used in the robotic vision community. In Part 2 and Part 3 you will further investigate the use of these algorithms to estimate the pose of the camera and to calibrate the helicopter model itself.

Report template

You do not need to include any text, derivation or figure that is not explicitly requested for, nor will you receive any points for it. Keep the report simple and provide only what is asked for. You can write the report using any convenient tool, as long as it contains the requested answers and the name of who you worked with (if you did not work alone). Also upload a **zip archive** of all your code.

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 numerical optimization. A summary of the necessary optimization theory is therefore given in Part 1. Alternatively, the booklet by Madsen, Nielsen and Tingleff (available on Blackboard) is a good introduction to non-linear least squares.

- Madsen, Nielsen, and Tingleff. Methods for non-linear least squares problems. 2004.

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 following dataset included in the zip:

- `video0000.jpg...video0350.jpg`: Undistorted image sequence.
- `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 $\mathbf{T}_{\text{platform}}^{\text{camera}}$.
- `detections.txt`: Marker detections. 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 and 0 otherwise, and (u_i, v_i) is the marker's pixel coordinates.
- `logs.txt`: Recorded angles from motor encoders. Each row contains a timestamp (in seconds), yaw, pitch and roll. The logs have been time-synchronized with the images.
- `platform_corners_metric.txt`: Metric coordinates of four points on the platform (Part 2).
- `platform_corners_image.txt`: Measured pixel coordinates of the above points (Part 2).

Note that the image locations of the markers have been extracted for you. Additionally, the images have had lens distortion removed, such that they satisfy a pinhole camera model with the provided intrinsic matrix $\mathbf{K}$. The zip contains helper code to load the dataset in Python and in Matlab.

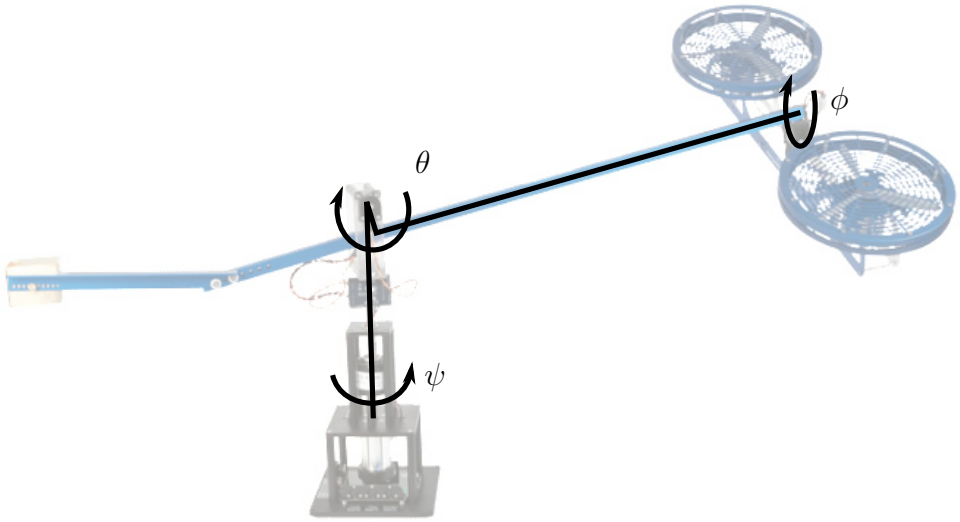

Figure 2: Quanser 3-DOF helicopter and its three degrees of freedom. The arrows indicate the direction of increasing yaw, pitch and roll.

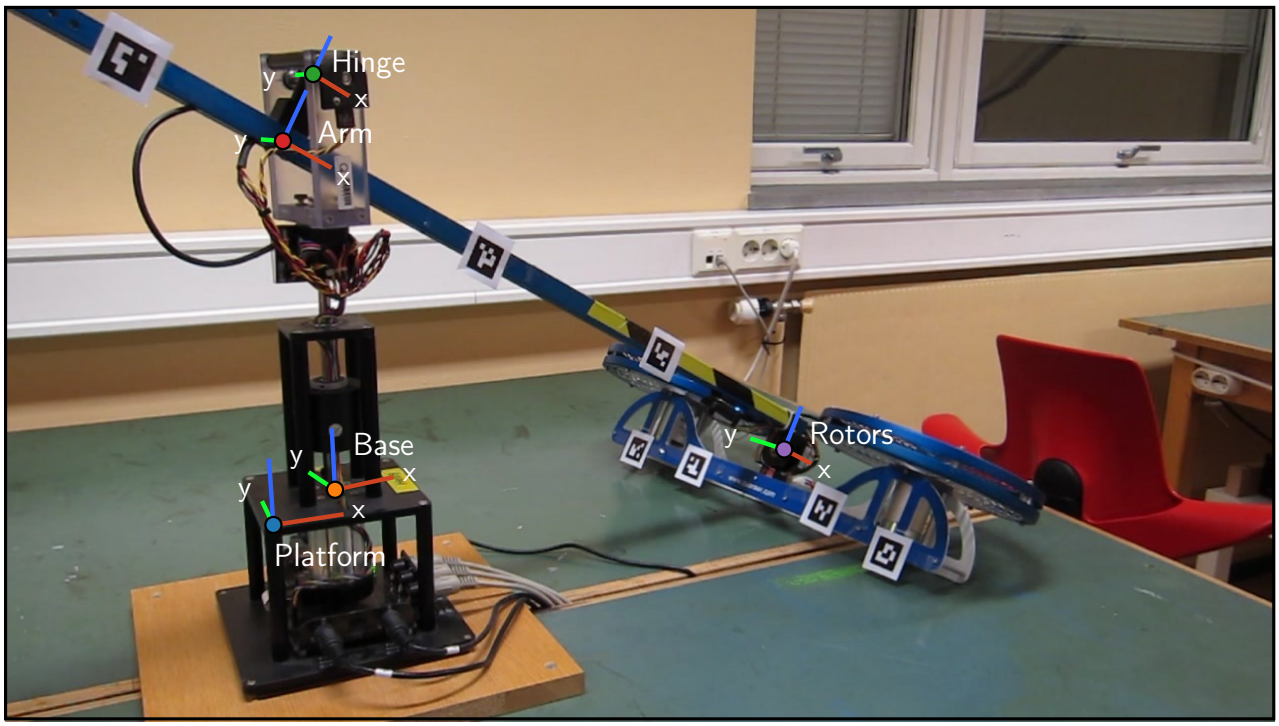

Figure 3: Helicopter coordinate frames

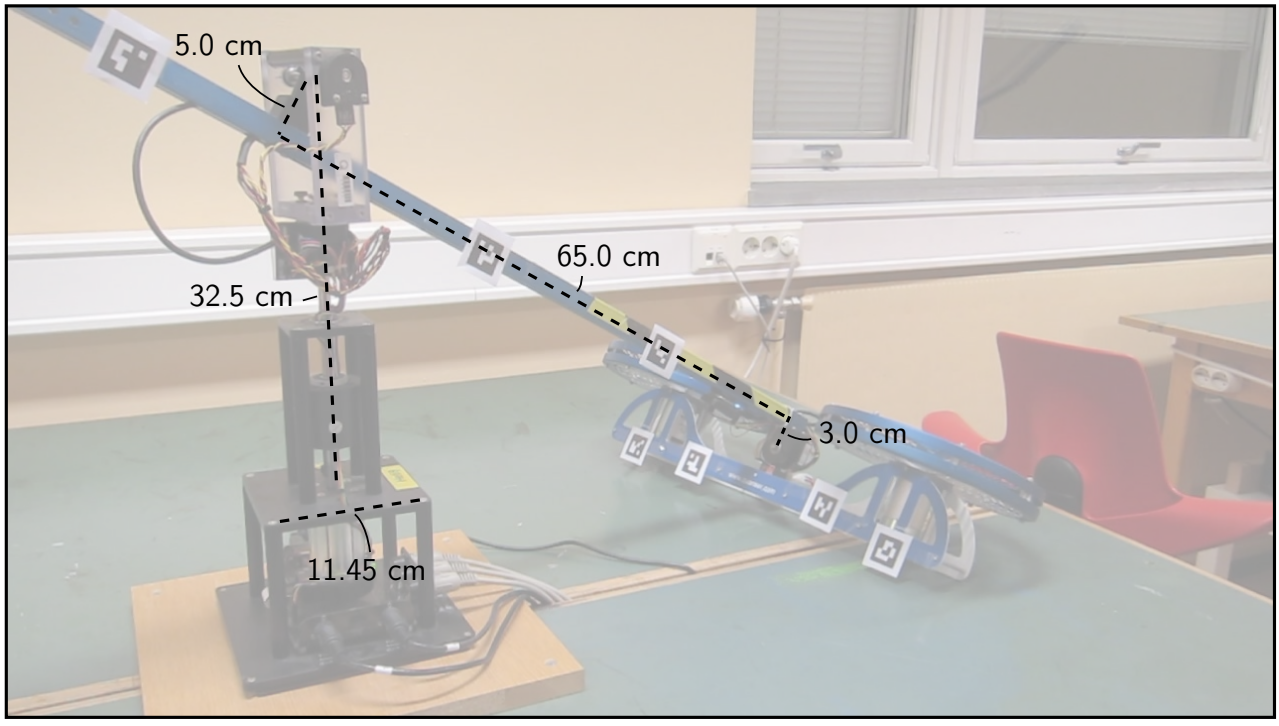

Figure 4: Helicopter dimensions

Part 1 Estimate the helicopter angles (10 points)

In this part you will implement an iterative algorithm to estimate the helicopter's yaw, pitch and roll. Compared with the DLT, this algorithm is more flexible and can easily incorporate constraints and other knowledge, such as motion dynamics and information from other sensors. The algorithm is based on the assumption that an optimal estimate of the helicopter angles will cause the projection of points in the helicopter model to fall close to their corresponding observations. 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 called the *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 computed from

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

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

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

Likewise, $\mathbf{u}_i$ is the observed image location of the i'th marker.

Note that although the markers have a square shape and could thereby 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. Although the field has recently discovered efficient global solvers for several interesting non-linear problems, these make strong assumptions about the underlying model. In absence of such a solver, the standard approach is to use a generic non-linear optimization method. These tend to be iterative, in that they start from an initial estimate and refine it over a number of iterations, so as to decrease the error. Such methods can at best guarantee convergence to a local minimum.

The Matlab Optimization Toolbox (notably `fmincon`) and IPOPT are well-tested software packages for solving generic 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 computational efficiency. The basis for these latter packages are the Gauss-Newton and Levenberg-Marquardt methods, which you will implement in this part. These methods start at an initial estimate and iteratively apply small corrections, obtained by solving a linear least squares problem constructed at the current estimate.

The Gauss-Newton method

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. Automatically determining the step size is a key part of a Gauss-Newton implementation, but we will see that the Levenberg-Marquardt method simultaneously provides automatic step size control and improves several other issues.

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). The Levenberg-Marquardt method is a small modification that addresses both issues simultaneously. The difference is that the step size α is omitted (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*, and is determined by the following rules. 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 (e.g. divided by 3) before the next iteration. Otherwise, μ is increased (e.g. multiplied by 2) 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 (locally).

A typical initial value of μ is 10^{-3} times the maximum of the elements on the diagonal of $\mathbf{J}^T\mathbf{J}$. In practice it's also a good idea to add a *termination condition* to prevent unnecessary iterations. A common choice is to stop when the change in the parameter vector between two successive iterations is less than a desired precision, measured e.g. using the norm of the accepted step $||\boldsymbol{\delta}^{\text{LM}}||$.

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 in Python, 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.

For this project, the most straightforward option may be to use the finite difference approximation. For a function of a scalar parameter, the central finite difference approximation of its partial 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. The best value depends on the scale of each parameter. In Part 1, the value 10^{-5} seems to work fine.

Task 1.1 (1 point)

In order to apply the described methods to our problem, we need to define the residuals $r_i(\mathbf{p})$. At a first glance, you may be tempted to define these as the norm of the vector differences,

$$\mathbf{r}(\mathbf{p}) = \begin{bmatrix} ||\hat{\mathbf{u}}_1(\mathbf{p}) - \mathbf{u}_1|| & \cdots & ||\hat{\mathbf{u}}_n(\mathbf{p}) - \mathbf{u}_n|| \end{bmatrix}^T, \tag{13}$$

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,

$$\mathbf{r}(\mathbf{p}) = \begin{bmatrix} (\hat{u}_1(\mathbf{p}) - u_1) & \cdots & (\hat{u}_n(\mathbf{p}) - u_n) & (\hat{v}_1(\mathbf{p}) - v_1) & \cdots & (\hat{v}_n(\mathbf{p}) - v_n) \end{bmatrix}^T. \tag{14}$$

Mathematically, both choices define the exact same objective function (the sum of squared residuals), but give rise to very different linearizations. Speculate on why the linearization produced by the first choice might be problematic.

The Gauss-Newton method can be derived generally for vector-valued residuals without this seemingly ad-hoc concatenation. The general formulation allows e.g. each 2D pixel error to be weighted by its 2×2 inverse covariance matrix (see Szeliski 6.1.1), which is used in the previously mentioned software packages. However, this is more cumbersome to implement in Python and Matlab. If the residuals are unweighted there is no difference with concatenation.

Task 1.2 (1 point)

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

Finish the implementation of `residuals` and run the `part1` script. It should print the residuals on the first image using the optimal angles. Check that they are small (within ± 10 pixels) and include these numbers in your report.

Note that not every marker is detected in each image, so some of the residuals may be invalid. Instead of returning only the valid residuals, it can be useful to keep the length of $\mathbf{r}$ the same for each image, and handle invalid residuals by multiplying the corresponding entries of $\mathbf{r}$ by 0 before returning the array. The hand-out code provides a vector called `weights` which you can use to achieve this.

Task 1.3 (2 points)

Write a function that implements the Gauss-Newton method. See the hand-out code for some tips. Modify the `part1` script to run the method up to image number 86, using a fixed step size of $\alpha = 0.25$ and 100 iterations. The `part1` script should generate a figure showing your estimated angles against the motor encoder logs (the helicopter's *trajectory*). Include only this figure in your report.

Task 1.4 (1 point)

Run Gauss-Newton beyond image number 86. In Matlab or Python, you should see a warning indicating that $\mathbf{J}^T\mathbf{J}$ is singular (not invertible). Explain why this happens here and when it can happen in general.

Task 1.5 (3 points)

Write a function that implements the Levenberg-Marquardt method.

Initialize $\psi = \theta = \phi = 0$ and run the method on only the first image. Use a maximum of 100 iterations and print the angle estimates and the value of μ in each iteration. What happens with μ? How many iterations does it take before the estimates stabilize to 0.001 radians precision?

Finally, modify the `part1` script to run Levenberg-Marquardt on the entire image sequence. Include the plot of the estimated angles against the motor encoder logs.

Task 1.6 (1 point)

If your Levenberg-Marquardt implementation is correct, you will no longer get the above warning. Why? Is this a good way to handle the underlying issue or can you imagine a better strategy? (One or two sentences is an acceptable answer).

Task 1.7 (1 point)

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 point correspondence (when we ignore the remaining three corners of the marker). Argue that there then exists up to two physical configurations (ψ, θ) that are local minimizers of the reprojection error (i.e. consistent with the marker observation).

You may include a sketch to support your argument. (A photograph of a paper drawing is fine).

Part 2 Estimate the platform pose (5 points)

In Part 1, the transformation $\mathbf{T}_{\text{platform}}^{\text{camera}}$ was given to you. Here you will investigate 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`.

Task 2.1 (2 points)

Estimate $\mathbf{T}_{\text{platform}}^{\text{camera}}$ 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 both sets of image locations. Also compute their reprojection errors and explain why the reprojection errors of (a) should all be exactly zero, and why the reprojection errors of (b) are all, except for one point, non-zero.

Task 2.2 (2 points)

Estimate $\mathbf{T}_{\text{platform}}^{\text{camera}}$ by minimizing the sum of squared reprojection errors with respect to $\mathbf{R}$ and $\mathbf{t}$, using Levenberg-Marquardt. Compare the resulting reprojection errors with those in the previous task.

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. 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. The reference rotation $\mathbf{R}_0$ may be obtained from the linear estimate. This parameterization suffers from gimbal lock in general, but for small angles it is fine. With three additional parameters for $\mathbf{t}$, you obtain a minimal (six-dimensional) parameterization of $\mathbf{T}_{\text{platform}}^{\text{camera}}$.

Task 2.3 (1 point)

Suppose you only have three point correspondences instead of four. Show that there can then be more than one physically plausible transformation $\mathbf{T}_{\text{platform}}^{\text{camera}}$ that minimizes the reprojection error, by finding a specific example of a different local minimum for this particular dataset. Include the 4×4 matrix in your report along with a short description of how you found it.

Part 3 Calibrate the model via batch optimization (10 points)

> This part is much more challenging and requires strong programming and debugging abilities, along with a good amount of self-direction. We do not expect everyone to complete this part.

The helicopter model will not perfectly match the physical reality. Besides inaccurate measurements of the various lengths and the 3D marker locations, the structural assumptions themselves 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 of this is an inaccurate estimate of the angles.

This can be addressed by performing *batch optimization*, whereby the static aspects of the model are jointly estimated with the variable degrees of freedom. This is similar to camera calibration, where the intrinsics are estimated from images from different viewpoints, under the assumption that only the camera pose varies from image to image, while the intrinsics are shared. Likewise, we assume that the helicopter model has a set of static parameters that are shared between the images. If the model is not over-parameterized, and if the helicopter undergoes sufficient motion over many images, then there should be a unique set of parameters that minimize the sum of reprojection errors over all the images.

Suppose that the helicopter model has m static parameters. The angles can change in each image, so for a sequence of length l there are $3l$ dynamic parameters, giving a total of $m + 3l$ optimization variables. This is a much larger optimization problem than before, and while you could use your methods from Part 1, they will be prohibitively slow due to the need to invert $\mathbf{J}^T\mathbf{J}$, which is now roughly a 1000×1000 matrix if you use all the images. However, the problem turns out to have a similar *bipartite* structure as the *bundle adjustment* problem (Szeliski 7.4), and can be solved much faster by exploiting sparsity.

In particular, notice that dynamic variables from different images are independent; adjusting the angles (ψ, θ, ϕ) for one image does not affect the reprojection error in a different image. This results in a sparse Jacobian and consequently a sparse matrix $\mathbf{J}^T\mathbf{J}$. If we partition and order the variables appropriately, then $\mathbf{J}^T\mathbf{J}$ will have a $3l \times 3l$ block, corresponding to the dynamic variables, which is block-diagonal with blocks of size 3×3. Inverting a block-diagonal matrix is easy, as you simply invert each block independently. The *Schur complement* lets you exploit this fact for solving the original system, see e.g. the Wikipedia entry (link) or Hartley and Zisserman A6.3.1.

Tips:

- Print the objective function value and make sure that it decreases in each iteration.
- When computing the Schur complement it's very easy to forget the minus sign in the normal equations: $\mathbf{J}^T\mathbf{J}\boldsymbol{\delta} = -\mathbf{J}^T\mathbf{r}$.
- Don't store or operate with the $3l \times 3l$ block as a dense matrix; store the individual 3×3 blocks and simplify the computations involving them.
- Add a damping term to the diagonal of each 3×3 block, as well as the $m \times m$ block.
- (For Python users) Don't create a reference where a copy was intended.

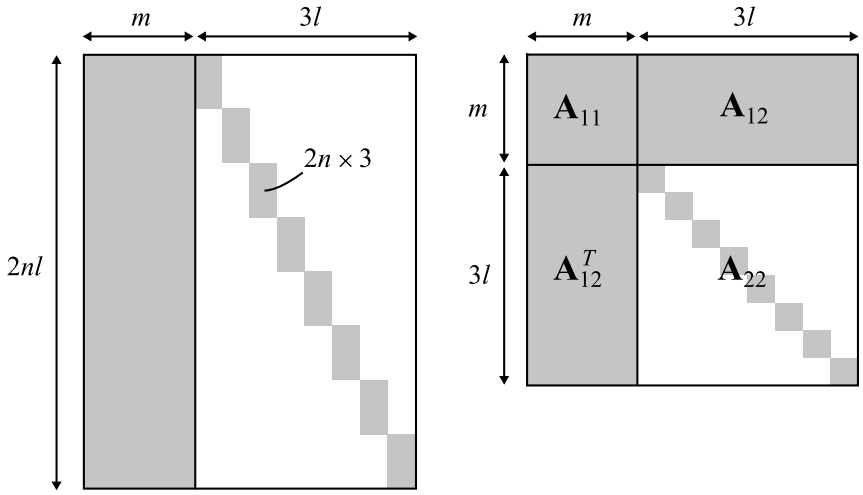

Figure 5: Form of the Jacobian $\mathbf{J}$ and the approximate Hessian $\mathbf{A} = \mathbf{J}^T\mathbf{J}$ for $l = 8$ images and m static parameters. The shaded rectangles indicate blocks of possibly non-zero entries, while the unshaded areas are all zeros. Note that the proportions here are not indicative of the matrices that you will obtain in your implementation; when you use all the available images, the $3l \times 3l$ block-diagonal matrix will be much bigger than the $m \times m$ static variable block.

Task 3.1 (9 points)

To begin with, let the five lengths indicated in Fig. 4 and the 3D marker coordinates be optimization variables rather than constants. This gives $m = 5 + 3n$ static variables, with $n = 7$ being the number of markers. Implement batch optimization in a new script and estimate the static model parameters. You will need a reasonable initial estimate of all the variables. You may use the inaccurate helicopter model and estimated trajectory from Part 1 for this. Modify the Quanser class to use the new lengths and marker coordinates, and rerun the `part1` script. Include all the generated figures in your report.

While the above parameterization will greatly reduce the error, it still has some room for improvement. Define an even more general helicopter model by letting the three rotation axes (yaw, pitch and roll) be fully parameterized. To avoid over-parameterization, each rotation axis should have four static variables, corresponding to two rotational and two translational degrees of freedom. (The third rotation is given by ψ, θ or ϕ, and any translation *not* perpendicular to the rotated axis is redundant). Repeat the batch optimization and run the `part1` script with the improved model. Include all the generated figures.

Task 3.2 (1 point)

With the second, more general, model from above, you should be able to achieve a mean reprojection error below 0.2 pixels. However, you may still observe a small biased error between the encoder logs and the vision estimates. The magnitude of the error varies smoothly over the trajectory, and is therefore not explained by potential synchronization errors (i.e. a horizontal and vertical offset). What could be the source of this remaining error?