Bundle adjustment

Contents

Much like KLT, bundle adjustment is a nonlinear optimization. In the previous exercise, we implemented the full optimization from scratch. In this exercise, we will simplify this task by outsourcing gradient calculation and descent to Matlab using the `lsqnonlin` command.

1 Preliminaries

1.1 Outline of the exercise

Every time a known landmark is observed, the image point of the observation provides information which could be used to refine both the landmark position estimate and the pose estimates of all frames that observed it. Incorporating this information in real time is hard, so we have not attempted this in any previous exercises or the suggested VO pipeline. In this exercise we will still not quite solve the problem of real-time incorporation, but we will see how all observations can be used to refine the trajectory estimate and the map off-line, i.e., once all data has been collected. This is typically done with an algorithm called *Bundle Adjustment* (BA). In BA (whose name derives from *bundles of light being adjusted*), trajectory and map estimate are refined in a way that minimizes the reprojection error, that is, the distance between where a landmark is observed on an image and where it should be observed according to the estimated geometry. BA is a special case of non-linear least-squares (NLLS) optimization.

In this exercise, we will see how BA can be formulated as a NLLS problem, and solve it using Matlab's `lsqnonlin` function. NLLS optimization is a tool that can be applied to a wide variety of problems - any major programming language should also have a library for this. This exercise should give you a lot of insight into how BA works, and should on the other hand make you familiar with `lsqnonlin` (or your programming language's equivalent), which is a handy tool for many other problems. Incidentally, it can be used (and we will use it) in the evaluation of BA. To evaluate the effectiveness of BA, we compare the adjusted estimate with the ground truth. However, ground truth and estimate vary in scale due to the unknown scale in monocular vision. Furthermore, any rotational error at the beginning of the estimate can rotate the estimated trajectory with respect to the ground truth. Thus, to evaluate how much the estimate resembles ground truth, we will first align the estimate to the ground truth, which can also be achieved using NLLS optimization. Thus, the outline of the exercise is as follows:

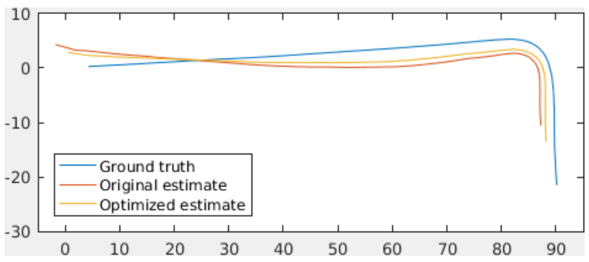

Figure 1: The first 150 frames of KITTI 00 describe an L-shaped trajectory with a long first segment and a right turn followed by a shorter segment. Here we can see that our reference VO exhibits scale drift, as the first part of the aligned estimate is longer than the ground truth while the second part is shorter. Even though we limit our bundle adjustment to 20 iterations, we can see how it improves the accuracy of the estimate, by reducing and straightening the first part and enlarging the second. See the attached animated gif for an even better illustration of this.

First, we implement estimate-to-ground-truth alignment, since this is an easy non-linear least-squares problem. Then, we will apply `lsqnonlin` in its simplest form to a small BA problem, and see how this already reaches the computation limits of a typical laptop. We will then see how we can exploit knowledge of the problem to increase the efficiency of `lsqnonlin`. Finally, we will apply our optimized BA to a larger problem and see how it improves the accuracy of the trajectory estimate.

1.2 Provided code

As usual, we provide you with skeletal Matlab code (`main.m`) with a section for each part of the exercise. Your job will be to implement the code that does the actual logic. We also provide the functions stubs with some comments about the input and output formats, so if these are not clear from this PDF, they should be clear from the function stubs. *As usual, you do not need to reproduce the reference outputs exactly.* Note that you will be using code from previous exercises - you may use your own code, but it's probably less hassle if you use our reference implementations.

1.3 Conventions

Pose transformations between frames A and B are denoted with rotation matrix and translation vector R_{AB} and t_{AB} such that the origin of B expressed in A is at t_{AB} and the (x, y, z) unit vectors of frame B expressed in frame A are the columns of R_{AB}. With this, a point p_B expressed in B can be expressed in A as follows:

$$p_A = R_{AB} \cdot p_B + t_{AB}. \tag{1}$$

The inverse transformation is given by $R_{BA} = R_{AB}^T$ and $t_{BA} = -R_{AB}^T \cdot t_{AB}$. W denotes the world or global frame and C the camera frame. The camera looks in positive z direction, x points to the right and y down in the image. For trajectory alignment there are two "world frames": The frame in which the estimated trajectory is expressed, and the frame in which the ground truth lives. We call these V (for visual odometry) and G, respectively. Also, a prime ($'$) will indicate ground truth variables (points, poses).

1.4 Data format

Unless otherwise stated, collections of vectors are stored as matrices, with the vectors in the columns. For parts 2 and 3, the trajectory and map are represented with two column vectors, `hidden_state` and `observations`:

$$\text{hidden\_state} = \left[\tau_1^T, ..., \tau_n^T, P_1^T, ..., P_m^T\right]^T \tag{2}$$

$$\text{observations} = \left[n, m, O_1^T, ..., O_n^T\right]^T \tag{3}$$

$$O_i = \left[k_i, p_{i,1}^T, ..., p_{i,k_i}^T, l_{i,1}, ..., l_{i,k_i}\right]^T, \tag{4}$$

where τ_i is the twist vector (see Section 2) representing the camera pose T_{WC} of the i-th frame, P_i the 3D position of the i-th landmark, n the number of frames, m the number of landmarks, k_i the number of landmarks observed in the i-th frame, p_i the 2D position of the keypoint corresponding to the i-th observed landmark (in (row, col) and **not** (x, y) of the image - you might need to flip this) and l_i the index of the corresponding landmark. For example, $p_{i,j}$ describes the observation of the landmark with position $P_{l_{i,j}}$ in the image of the frame described by τ_i.

2 Part 1: Trajectory alignment

In the first part, we apply non-linear least-squares (NLLS) optimization to align the original trajectory estimate to the ground truth. This is a problem that can actually be solved in closed form, but we will solve it with NLLS for practice. NLLS optimization solves the following problem (looks familiar? See previous exercise.):

$$\mathbf{x}^\star = \arg\min_{\mathbf{x}} \mathbf{e}(\mathbf{x})^T \mathbf{e}(\mathbf{x}) = \arg\min_{\mathbf{x}} \sum_i e_i(\mathbf{x})^2 \tag{5}$$

where $\mathbf{e}(\mathbf{x})$ is a vector containing error terms $e_i(\mathbf{x})$ as coefficients. In the situation where some model parametrized by $\mathbf{x}$ (hidden state) needs to be inferred from observations $\mathbf{Y}$, and we can model what $\mathbf{Y}$ should be given $\mathbf{x}$, (5) can be used by setting

$$\mathbf{e}(\mathbf{x}) = \mathbf{f}(\mathbf{x}) - \mathbf{Y} = \begin{bmatrix} f_1(\mathbf{x}) - Y_1 \\ f_2(\mathbf{x}) - Y_2 \\ \dots \end{bmatrix} \tag{6}$$

where for each observation Y_i, the model function $f_i(\mathbf{x})$ describes what that observation should be according to the model parametrized by $\mathbf{x}$. For aligning the trajectory estimated by the VO, $\mathbf{p}_V^{(i)}$, to the ground truth trajectory, $\mathbf{p}_G'^{(i)}$, we assume that there is some similarity transformation $S_{GV} = \begin{bmatrix} s_{GV} \cdot R_{GV} & t_{GV} \\ 0 & 1 \end{bmatrix}$ such that

$$\begin{bmatrix} \mathbf{p}_G'^{(i)} \\ 1 \end{bmatrix} \sim S_{GV} \begin{bmatrix} \mathbf{p}_V^{(i)} \\ 1 \end{bmatrix} \quad \text{or} \quad \mathbf{p}_G'^{(i)} \sim s_{GV} \cdot R_{GV} \mathbf{p}_V^{(i)} + t_{GV} \tag{7}$$

Note that S_{GV} is similar to what we have previously used for 3D transformations, except that it additionally contains a scale factor s_{GV} which models a change in scale. If the VO estimate were perfect, an S_{GV} would exist that would solve (7) exactly. However, since the estimate is faulty, we look for S_{GV} which minimizes the difference between $\mathbf{p}_G'^{(i)}$ and $S_{GV}\mathbf{p}_V^{(i)}$ (henceforth a shorthand for the top three rows of $S_{GV} \begin{bmatrix} \mathbf{p}_V^{(i)} \\ 1 \end{bmatrix}$). In particular, we can use (5) and (6) with $\mathbf{Y} = (\mathbf{p}_G'^{(1)T}, \mathbf{p}_G'^{(2)T}, \dots, \mathbf{p}_G'^{(n)T})^T$ and $\mathbf{f}(\mathbf{x}) = (S_{GV}\mathbf{p}_V^{(1)T}, \dots, S_{GV}\mathbf{p}_V^{(n)T})^T$. Note that each point in the trajectory will have three coefficients in $\mathbf{e}$, one for each dimension in 3D.

We will now solve this using `lsqnonlin`. `lsqnonlin` takes as arguments an initial guess for the model parameters $\mathbf{x}$, in form of a vector, and the error function $\mathbf{e}(\mathbf{x})$. It will then tweak $\mathbf{x}$ using sophisticated gradient descent algorithms until $\mathbf{e}(\mathbf{x})^T\mathbf{e}(\mathbf{x})$ reaches a (local!) minimum. To use this for trajectory alignment, we need to express parameters describing S_{GV} as a vector. Since S_{GV} needs to satisfy constraints to be valid (e.g. R_{GV} needs to be a valid rotation matrix), we cannot simply reshape it to a vector and pass it to `lsqnonlin`, since `lsqnonlin` might violate these constraints. Instead, we should choose a minimal representation that makes it impossible to violate these constraints. This can be done in a couple of ways, and we choose to represent rotation and translation with a twist vector, after separating scale. To convert between a homogeneous matrix $\begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix}$ and twist vector τ, we provide the functions `HomogMatrix2twist` and `twist2HomogMatrix`. Then, we can set $\mathbf{x} = \begin{bmatrix} \tau_{GV} \\ s_{GV} \end{bmatrix}$. With this, conversion from $\mathbf{x}$ to S_{GV} should be straightforward. The

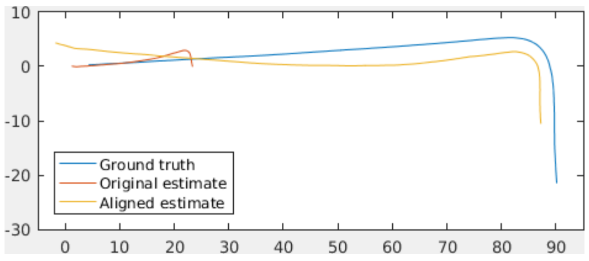

Figure 2: Original estimate and estimate aligned to ground truth.

inverse can be e.g. achieved by looking at the determinant of the top left 3×3 block of S_{GV}, but won't actually be necessary in this exercise. For the initial guess, you can set $\mathbf{x} = \begin{bmatrix} \tau(I_{4\times4}) \\ 1 \end{bmatrix}$.

Complete the function `alignEstimateToGroundTruth` to return the aligned $S_{GV}\mathbf{p}_V^{(i)}$, given $\mathbf{p}_G^{\prime(i)}$ and $\mathbf{p}_V^{(i)}$. In your call to `lsqnonlin`, set the option `Display` to `iter`. You should get the results as shown in Fig. 2 and a final squared error of around $3000 m^2$.

Hint: `lsqnonlin` needs a function that depends only on $\mathbf{x}$, but $\mathbf{e}(\mathbf{x})$ also depends on $\mathbf{p}_G^{\prime(i)}$ and $\mathbf{p}_V^{(i)}$. We suggest that you first write a function `alignmentError` that expresses $\mathbf{e}(\mathbf{x}, \mathbf{p}_G^{\prime(i)}, \mathbf{p}_V^{(i)})$, then bind the input arguments $\mathbf{p}_G^{\prime(i)}$ and $\mathbf{p}_V^{(i)}$ to the given values. In Matlab this can be done by writing, in the body of `alignEstimateToGroundTruth`: `error_function = @(x) alignmentError(x, pp_G_C, p_V_C);` . Then, `error_function` can be passed as first argument to `lsqnonlin`.

3 Part 2: Small bundle adjustment

Now that you have applied `lsqnonlin` to a simple problem, let us apply it to bundle adjustment (BA). BA fits the formulation in equations (5) and (6). In BA, $\mathbf{x}$ represents the frame poses and landmark positions, $\mathbf{Y}$ the 2D coordinates of each landmark observation in each image, and $\mathbf{f}(\mathbf{x})$ the 2D coordinates that should be observed according to the model defined by $\mathbf{x}$. As we will see, a naive implementation of bundle adjustment with `lsqnonlin` is rather inefficient, and so we only start with the first four frames. For your convenience, we have pre-packaged the problem data in a way that is appropriate for `lsqnonlin`, see Section 1.4.

Implement `runBA`, where, given x and O, you formulate $\mathbf{Y}$ and $f(\mathbf{x})$ in a way that minimizes, for each observation of each landmark, the distance between the projection of the 3D landmark onto the image plane and the actual observation of that landmark (the so-called **reprojection error**). Limit lsqnonlin to 20 iterations. With this small problem (four frames only), it is hard to notice improvement with respect to the initial estimate. Our output is shown in Fig. 3. Essentially, as long as something has changed in your map without destroying it, you are probably fine. We will evaluate the BA performance in the final part of this exercise.

Some hints:

- It is normal if this function executes in a couple of minutes. We will see how to make it more efficient in the next part.
- There are plenty of sources for bugs in this exercise. We strongly encourage you to make debug plots. For instance, you should verify that landmark observations and projections are always very close to each other (this is of course already the case in the output given by the VO - bundle adjustment is about making these distances even smaller).
- You should have $\mathbf{Y} = \left[p_{1,1}^T, ..., p_{1,k_1}^T, ..., p_{n,1}^T, ..., p_{n,k_n}^T\right]^T$. In particular, each landmark observation will have two coefficients in $\mathbf{e}(\mathbf{x})$, one for each dimension in 2D.
- To get some feedback during execution we recommend to also here set the `lsqnonlin` option `Display` to `iter`.

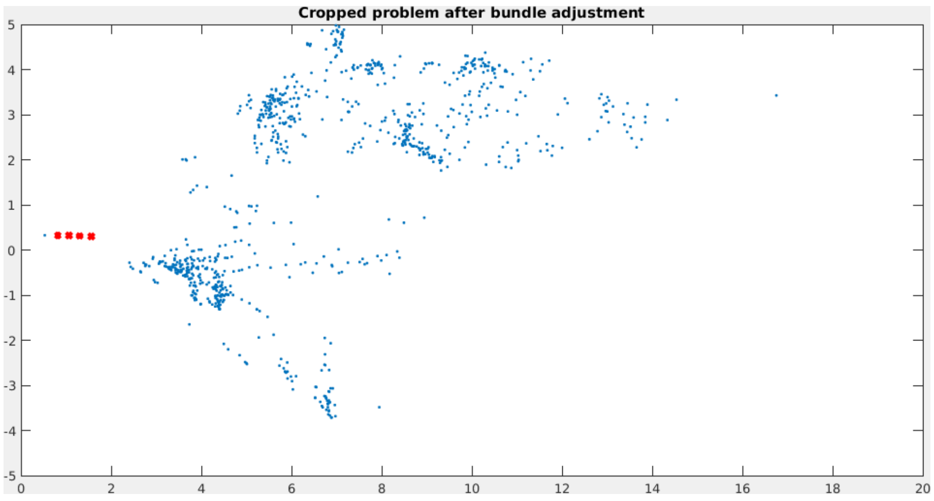

Figure 3: The outcome of running bundle adjustment on the first four frames.

- Same as in the previous part of the exercise, we recommend to write an error function which takes both hidden and observed state, and then bind the observed state to the values provided as input to your function (O).

4 Part 3: Determining the Jacobian pattern

As you should see, even with as simple a map as one with only four frames, our naive implementation of bundle adjustment (BA) with `lsqnonlin` takes a lot of time. This is because Matlab spends a lot of time numerically figuring out the Jacobian

$$J = \begin{bmatrix} \frac{\partial e_1}{\partial x_1} & \frac{\partial e_1}{\partial x_2} & \cdots \\ \frac{\partial e_2}{\partial x_1} & \frac{\partial e_2}{\partial x_2} & \\ \cdots & & \ddots \end{bmatrix} \tag{8}$$

which is needed to calculate the gradient, and thus the direction in which `lsqnonlin` will change $\mathbf{x}$ in order to decrease the error most effectively. Essentially, Matlab takes every single coefficient of x and slightly changes it to see what happens to every coefficient of e (**numerical differentiation**[1]). Thus, `lsqnonlin` (and other libraries for NLLS optimization) can be significantly accelerated by providing them with an expression for the Jacobian[2]. Since the Jacobian is quite challenging to define in our case (especially since our model involves rotations) we will not provide a Jacobian function to lsqnonlin in this exercise. What we can do, however, is tell it which coefficients of the Jacobian are zero. A Jacobian coefficient J_{ij} is nonzero only if the i-th coefficient of $\mathbf{e}(\mathbf{x})$ is affected by the j-th coefficient of $\mathbf{x}$ ($\frac{\partial e_i}{\partial x_j} \neq 0$). In our case, coefficients of $\mathbf{e}$ correspond to reprojection errors while coefficients of $\mathbf{x}$ correspond to camera poses and landmark positions. But as you can imagine, not every camera pose and landmark will affect every reprojection error. In fact, a single reprojection error depends only on the pose of the camera which observes it and the corresponding landmark! So, we can significantly reduce the numerical differentiation effort of `lsqnonlin` by telling it which coefficients of the Jacobian to skip.

In `runBA`, create a pattern matrix M with $M_{ij} = 1$ if the i-th coefficient of $\mathbf{e}(\mathbf{x})$ is affected by the j-th coefficient of $\mathbf{x}$ and $= 0$ otherwise. Then, pass M to `lsqnonlin` with the option `JacobPattern`.

[1] https://en.wikipedia.org/wiki/Numerical_differentiation

[2] In fact, for some languages (for example C++) there is a thing "between" numerical differentiation and having to provide the Jacobian called automatic differentiation. Check it out, it is a pretty cool idea: https://en.wikipedia.org/wiki/Automatic_differentiation . It provides a nice compromise between the tediousness of manually calculating the Jacobian and the slowness of numerical differentiation. Unfortunately, it is not available in Matlab.

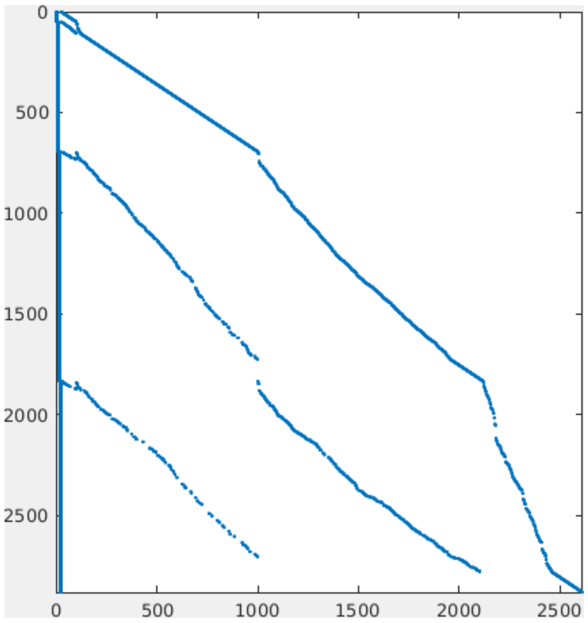

Figure 4: The Jacobian pattern of the small bundle adjustment problem.

`lsqnonlin` will now estimate J_{ij} only when $M_{ij} = 1$, which in the case of BA massively speeds up its execution. Hints:

- You can visualize M with the Matlab command `spy`. This should look like in Fig. 4.
- You don't need to do any math (apart from indexing) to determine whether M_{ij} should be zero or not. Just think about which parts in $\mathbf{x}$ affect the 2D position of the keypoint corresponding to a given landmark in a given frame.
- Don't forget that each observation has two coefficents in $\mathbf{e}$.
- Important: You probably don't have enough RAM to represent M densely in Part 4. Instead, use `sparse`. Your code will be even faster if you pre-allocate the memory for M using `spalloc`. It is not hard to predict how many non-zero elements M will have.

5 Part 4: Larger bundle adjustment and evaluation

Now that we have accelerated bundle adjustment (BA), we can run it on a larger problem. Execute the parts of `main.m` which run the BA function on the first 150 frames of KITTI (`hidden_state`, `observations`). This will take a couple of minutes to calculate. The map visualization of before and after BA should look like in the gif attached to the exercise. Finally, we can take the refined trajectory estimate and align it to the ground truth. With the bundle-adjusted estimate, the final alignment error should only be around $1500m^2$, and the plot which compares it to the pre-BA alignment should look like in Fig. 1.