Exercise 4 The Starry Night Dataset and Gauss-Newton optimization of poses ("point-cloud tracking" problem)

Guillermo Gallego

Here are some suggestions to guide you on the way to completing exercise 4. The goal is to attack the problem in a progressive way, increasing complexity.

- Understand the **data** provided by inspecting / **visualizing** it.
  - Print the variables in the data. See what ranges / orders of magnitude they have. Try to figure out their units, if these are not clear.
  - Visualize the data:
    * Plot the 3D points (do they look like the points in the problem statement?)
    * Plot the measured image points at each time step, like in a movie. Plot points from the left and right cameras on the same image plane, to see if they are approximately on the same horizontal line (epipolar lines).
    * Make the requested plot of the visible landmarks. Does it agree with what you see in the previous "movie" of image points?
    * Implement the measurement model and visualize the image point predictions vs. the actual measurements. Are the measured and predicted points close to each other and at a reasonable distance with respect to the variance given for the image points?
  - Visualize the DOFs of the ground truth poses (3DOFs for translation and 3DOF for rotation, e.g., rotation vector parametrization)
    * Visualize also the error between the ground truth poses and the poses obtained after implementing dead reckoning (the motion model using the IMU)
- **Incremental approach** to the Optimization problem
  - Write optimization code to obtain the best camera poses using only error terms from the stereo measurements (image points), not from the IMU data. To this end, choose a good test interval of images with enough image points (say frames $k_1 = 1000$ to $k_2 = 1049$), so that the Hessian is invertible (well-conditioned optimization problem).
  - Every once in a while try to profile the code to see where the bottlenecks are, and try to remove them without changing the functionality (i.e., without seriously affecting accuracy). Fast code makes it easier to test, and therefore, to improve.
  - For the optimization, start with a simple parametrization (e.g., rotation vector and translation). Do not worry about the Lie-flavored Gauss-Newton method (perturbation on the left, Chapter 8 in the book, etc.) until the chosen optimization routine (lsqnonlin in Matlab, scipy.optimize.least_squares in Python) is working with the simple parametrization.
  - When testing the optimization method (e.g., lsqnonlin), start the optimization near the expected minimizer by using the ground truth poses.
    * Visualize the results to check: Do the returned poses (minimum of the objective function) produce predicted stereo image points that are very close to the actual measurements? Visualization is also good during prototyping and checking the results, not only to see the input data.

    - Then, implement dead reckoning and check that both initializations (ground truth, dead reckoning) lead to the same minimizer of the non-linear least squares objective function.
  - Add the IMU (odometry) error terms. Stack all errors into a single residual vector and optimize.
  - Use the covariances of the image points and the IMU to weigh the error terms appropriately. We are using diagonal covariance matrices; so they are easy to implement.
  - After the optimization, compute the Hessian $\texttt{H} \approx \texttt{J}^\top \texttt{J}$ ($\texttt{J}$ is the Jacobian of the error vector with respect to the parameters – it should be returned by the optimization function) and invert it (better using sparse matrices), to obtain the covariance matrix at the minimizer. The diagonal of this covariance matrix are the variances used in uncertainty plotting (like in a previous exercise). Try to visualize the sparsity patterns of $\texttt{J}$ and $\texttt{H}$ (spy command in Matlab).
  - Do not use analytical derivatives (until the very end, once everything is working). Use numerical derivatives, automatic differentiation, or rely on Matlab, Python optimization functions to compute them internally. You should just care about computing the error terms, not their derivatives.
  - Only if there is time, I encourage you to code the Lie-flavored Gauss-Newton method yourself.

- Think how to **structure** your code to reutilize it and make it easy to write, understand and debug. Suggested functions:
  - Function to predict image points $\hat{\mathbf{y}}_k^j$ from the 3D points and candidate poses. Equations (3.4) and (3.5) in the assignment.
  - Function to plot the image points $\mathbf{y}_k^j$.
  - Function to dead reckon, i.e., to compute poses (Eq. (3.2)) given an initial pose and the IMU data (linear and angular velocities in Eq. (3.3.)). Basically, implement the motion model.
  - Function to convert a parameter vector ($6N \times 1$) into a set of poses (N rotation matrices and translation vectors)
    - Inverse function: convert a set of poses into a parameter vector. This can be used to convert the ground truth poses to a parameter vector that is passed to the optimization method.
  - Function to compute the error (vector) due to the stereo measurements: $\hat{\mathbf{y}}_k^j - \mathbf{y}_k^j$
    - Internally, it calls the function to predict image points $\hat{\mathbf{y}}_k^j$.
  - Function to compute the error (vector) due to the IMU data (motion model).
    - Internally, it may call the function to dead reckon.
  - Function that stacks the error vectors due to stereo and IMU data.
    - Internally, it calls the above two functions to compute the errors due to stereo data and to IMU data.
    - This is the function that is called by the optimizer.