Technische Universität München
Fakultät für Informatik
PD Dr. Slobodan Ilic

WS 19/20
Homework Sheet 1.2
November 21, 2019

Exercises in Tracking & Detection

Task 3 Pose refinement with non-linear optimization

Problem description The final task is to preform tracking of the camera in respect to the given 3D model. The test sequence is given in **images\tracking** folder.

As a first step, detect the object in the first image I_0 and compute an initial pose hypothesis using PnP and RANSAC from the previous exercise. After getting an initial pose hypothesis from the initial frame you need to do pose estimation for consecutive frames. This includes minimizing the re-projection error between 3D points on the model corresponding to the detected feature points in the previous image and the feature points detected in the current image. This will result in writing an objective function that should be minimized in terms of the camera pose (rotation and translation). Then those 3D points are projected into the current frame with the pose from the previous frame

First, you have to back-project SIFT features of the previous frame to the object by finding intersections with optical rays as in Homework 1.1. The previous frame is the one where the camera pose is already computed. For example at the start it will be the initial frame. The current frame is the one for which we want to estimate the camera pose. Then 3D points of back-projected SIFT features are projected into the current frame with the pose from the previous frame. Now you need to find matches between these projected SIFT features and SIFT features of the current frame. Finally, you have to minimize the reprojection error between the matches. In Eq. 1 $\mathbf{M_{i,t}}$ denotes the 3D point of back-projected SIFT feature of the previous frame and $\tilde{\mathbf{m}}_{i,t}$ denotes the 2D SIFT feature of the current frame

We are now given an initial pose $[\mathbf{R}_0, \mathbf{T}_0]$, the intrinsic matrix $\mathbf{A}$ and the point correspondence pairs between image points $\tilde{\mathbf{m}}_{i,t}$ and the 3D coordinates $\mathbf{M}_{i,t}$ of the 2D feature points. In order to compute the current camera pose $[\mathbf{R}_t, \mathbf{T}_t]$, we formulate an energy function $\mathbf{f}_t$ and apply non-linear optimization tools. One possible formulation of $\mathbf{f}_t$ is as follows:

$$\mathbf{f}_t(\mathbf{R}_t, \mathbf{T}_t; \mathbf{A}, \mathbf{M}_{i,t}, \mathbf{m}_{i,t}) = \sum_i \|\mathbf{A}\left(\mathbf{R}_t\mathbf{M}_{i,t} + \mathbf{T}_t\right) - \tilde{\mathbf{m}}_{i,t}\|^2 \tag{1}$$

As explained in the lecture, the camera matrix used in (back-)projection is composed from the intrinsic parameters in the following manner:

$$\mathbf{A} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2}$$

We assume no non-linear distortion effect. While this is not true in practice, and will result in slightly perturb correspondences, it should be a good approximation for the purposes of this exercise.

Implementation tasks In this homework you are required to implement Iterative Reweighed Least Square(IRLS) method for non-linear optimization of the camera pose parameters in the provided tracking sequences. This method is version of Levenberg-Marquart optimization algorithm. IRLS takes care of the outliers during iterations of non-linear optimization using robust estimator. The pseudo code for IRLS algorithm is given below as Algorithm 1.

Algorithm 1 IRLS: Iteratively re-weighted least squares.

Require: Data $\mathbf{x} = \{\boldsymbol{x}\}$ with $|\{\boldsymbol{x}\}| = N$, Initial parameters $\boldsymbol{\theta}_0$, Iterations T, Update threshold τ
Ensure: Solution $\boldsymbol{\theta}$

$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_0$
$t \leftarrow 0$
$\lambda \leftarrow 0.001$
$u \leftarrow \tau + 1$
for $t < T$ and $u > \tau$ **do**
  $\mathbf{e}_{2N\times 1} \leftarrow [d_u(\mathbf{x}, \boldsymbol{\theta})\ d_v(\mathbf{x}, \boldsymbol{\theta})]^T$ ▷ $\mathbf{e} = [e_1\ e_2\ \ldots\ e_{2N}]^T$
  $\sigma \leftarrow 1.48257968\ \text{MAD}(\mathbf{e})$ ▷ compute scale.
  $\mathbf{W}_{2N\times 2N} \leftarrow diag[\ldots, w(e_i/\sigma; c), \ldots]$
  $E(\mathbf{x}, \boldsymbol{\theta}) = \sum_{i}^{2N} \rho(e_i)$
  $\mathbf{J} \leftarrow \mathbf{J}(\mathbf{e})$
  $\Delta \leftarrow -(\mathbf{J}^T\mathbf{W}\mathbf{J} + \lambda\mathbf{I})^{-1}(\mathbf{J}^T\mathbf{W}\mathbf{e})$ ▷ compute update
  if $E(\mathbf{x}, \boldsymbol{\theta} + \Delta) > E(\mathbf{x}, \boldsymbol{\theta})$ **then**
    $\lambda \leftarrow 10\lambda$
  else
    $\lambda \leftarrow \lambda/10$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta$
  end if
  $u \leftarrow \|\Delta\|_2$
  $t \leftarrow t + 1$
end for

We suggest to start the implementation of the above algorithm by trying to understand and implement the following parts:

a) **Energy Function** Implement the energy function **f** (denoted with $E(x, \boldsymbol{\theta})$) in `MATLAB` that takes as input the rotation parameters (given in Exponential Maps, you can use the function *rotationMatrixToVector*), the translation parameters $\mathbf{T}$, the intrinsic matrix $\mathbf{A}$ and the 3D-2D correspondences $\mathbf{M}_i, \mathbf{m}_i$. Three things you should know:

- $\mathbf{A}$ and $\mathbf{M}_i, \mathbf{m}_i$ are known, the only unknown parameters are camera rotations and translations and they have to be estimated.

- Although it is not explicitly written in Eq. 1, after projecting 3D points to the 2D image plane, one should always divide all the coordinates by the Z component in order to get x and y coordinates of the re-projected point. You can use the MATLAB function *worldToImage* to compute the re-projected points given corresponding 3D points, camera pose and intrinsics.

- 3D rotations have only 3 degrees of freedom. Optimizing directly on a 9-element rotation matrix $\mathbf{R} \in SO(3)$ is not only unnecessary but it is also redundant. The exponential map $\mathbf{R}(\mathbf{v})$ comes handy in this case and we like you to use it.
  For rotation representation with exponential maps, you might find the Rodrigues

formula beneficial. Matlab makes it easy to convert to and back a rotation matrix to/from a Rodrigues vector. For further documentation, consult: `https://de.mathworks.com/help/vision/ref/rotationmatrixtovector.html`

b) **Jacobian computation.** The second step is to find the derivatives of the objective function and to create its Jacobian matrix. For double checking the Jacobians of its individual components, two things are helpful: 1) using MATLAB's Jacobian command to symbolically computing the matrix, 2) using finite differences as an approximation and checking for the proximity of the estimates to the analytical one. The choice of parameterization and derivative formulation for the rotational part is up to the implementation and you are generally free to choose any one. However, we suggest to first stick to the derivative of a rotation matrix $\mathbf{R}$ w.r.t. its exponential coordinates $\mathbf{v}$ as presented in *Guillermo Gallego and Anthony Yezzi (2014): A compact formula for the derivative of a 3-D rotation in exponential coordinates.* `https://arxiv.org/pdf/1312.0788.pdf` Let $\mathbf{R}(\mathbf{v}) = \exp([\mathbf{v}]_x)$ denote the rotation matrix as a function of the exponential coordinates $\mathbf{v} \in \mathbb{R}^3$. The operator $[\mathbf{v}]_\times$ turns the exponential coordinates $\mathbf{v}$ into a skew-symmetric matrix:

$$[\mathbf{v}]_\times = \begin{bmatrix} 0 & -v_3 & v_2 \\ v_3 & 0 & -v_1 \\ -v_2 & v_1 & 0 \end{bmatrix}$$

where scalar v_i is the i-th component of $\mathbf{v}$. Their main formula is equation (9) presented in their Result 2, and reads as:

$$\frac{\partial \mathbf{R}}{\partial v_i} = \frac{v_i[\mathbf{v}]_\times + [\mathbf{v} \times (\mathbf{I} - \mathbf{R})\mathbf{e}_i]_\times}{\|\mathbf{v}\|^2}\mathbf{R}$$

with $\boldsymbol{e}_i$ being the i-th vector of the standard basis in $\mathbb{R}^3$. $\mathbf{I}$ is the identity matrix as in $\mathbf{R}\mathbf{R}^T = \mathbf{I}$. For a proof and explanation see the reference.

c) **Outlier treatment** Due to SIFT correspondences the input might be corrupted by outliers. Therefore, one should take particular care of the outliers. This particularly becomes handy when frame-to-frame tracking has to be done. While there are many ways of doing that, we will employ a weighted non-linear optimization procedure in this exercise using robust norms (M-estimators) as shown in the lecture and oulined in the Algorithm1.

Let us think of a general energy:

$$E(\boldsymbol{\theta}) = \sum_{i=1}^{N} w_i d(\mathbf{x_i}, \boldsymbol{\theta})^2$$

where $\boldsymbol{\theta}$ are the optimized parameters, $\mathbf{x}$ are data points in domain Ω and d is an arbitrary distance function - in our case the re-projection error. We are always free to re-write E in terms of the residuals $\mathbf{e}^T = [\mathbf{e}_u^T \ \mathbf{e}_v^T]_{1\times 2N}$. :

$$E(\boldsymbol{\theta}) = \mathbf{e}^T\mathbf{e}.$$

Note that you can also alternate residual per u and per v coordinate in the final residual vector $\mathbf{e}$, that is up to you. Hence the gradient becomes:

$$\nabla E(\boldsymbol{\theta}) = 2(\nabla \mathbf{e})^T\mathbf{e}$$

giving us the Jacobian $\mathbf{J}_{2N\times 6} = \nabla \mathbf{e}$. From here, a typical Gauss-Newton follows:

$$(\mathbf{J}^T\mathbf{J})\Delta = -\mathbf{J}^T\mathbf{e}$$

with the typical update step:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \Delta.$$

By being a similar GN-family descent method, LM also admits the weighted variant. What is left is the determination of the weighting factors, which are given by the Tukey's bisquare M-estimator:

$$\rho(e) = \begin{cases} \frac{c^2}{6}\left(1 - \left(1 - \left(\frac{e}{c}\right)^2\right)^3\right), & \text{if } e \leq c \\ \frac{c^2}{6}, & otherwise \end{cases} \tag{3}$$

$$w(e) = \begin{cases} (1 - e^2/c^2)^2, & \text{if } e < c \\ 0, & otherwise \end{cases} \tag{4}$$

where $c = 4.685$ is used based on the assumption of variance-1 with 95% rate in the outlier rejection. $\rho(e_i)$ denotes the actual Tukey loss, whereas $w(e_i/\sigma)$ is the derivative for the i'th residual component e_i in $\mathbf{e} = [e_1\ e_2\ \ldots\ e_{2N}]^T$. $w(e_i/\sigma)$ is used to fill the weight matrix $W_{2N\times 2N}$ used in the update step which directly contribute to the gradient minimization. σ is computed as follows:

$$\sigma = 1.48257968\ \text{MAD}(\mathbf{e})$$

where MAD is the median absolute deviations: $\text{MAD}(\mathbf{e}) = median(|\mathbf{e}|)$.

Alternative way to GN/LM-update The Gauss-Newton procedure can also be written as a weighted least squares:

$$\Delta = \left[(\mathbf{J}^T\mathbf{W}\mathbf{J})\right]^{-1}(-\mathbf{J}^T\mathbf{W}\mathbf{e})$$

Instead of weighting the points/residuals, one might like to use this update directly. Note that, LM requires an evaluation of the actual function to compute the damping, whereas GN operates only on derivatives and this only requires w and not ρ.

Expected outcome The dataset consists of two sequences of images: validation and test. The validation sequence contains ground truth poses to verify your results. You will be given the code skeleton to help organise your solution however you are free to use your own code structure and visualisations.

We expect you to do the following:

- Implement the IRLS algorithm and perform tracking on validation and test sequences. You may implement the analytical Jacobian as described above but you can also use finite differences method or symbolic math toolbox. However, we don't accept any form of built-in solution for IRLS.
- Compute and save the trajectory of the camera in the world coordinate system for validation and test sequences.
- Compute and save per-frame visualization of predicted poses as projection of 3D bounding box onto the image plane (as in Homework 1.1) for validation and test sequences.

And, of course, we expect that you understand your code and can explain how it works as well as you can explain the theoretical aspects of your solution.

Base maximum grade for this exercise is 10 points. You can get 3 additional points by completing the following tasks:

- **2 points:** Implement a second different method to compute Jacobian.
- **1 point:** Convert your predicted poses for validation and test sequences to the trajectory format as described here : `https://vision.in.tum.de/data/datasets/rgbd-dataset/file_formats`.

  Compute the Absolute Trajectory Error (ATE) and Relative Pose Error (RPE) for the validation sequence and visualise the results. For that you can use tools from: `https://vision.in.tum.de/data/datasets/rgbd-dataset/tools`

  Send us your predictions for test and validation sequences in the trajectory format from above together with your code submission. For this task we are expecting the mean absolute translational error (from ATE) to be approximately less than $1cm$. In this task you should implement you own function to convert rotation matrix to quaternion.