Technische Universität München
Fakultät für Informatik
PD Dr. Slobodan Ilic

WS 18/19
Exercise Sheet 3
November 29, 2018

Exercises in Tracking & Detection

Task 1 Pose refinement with non-linear optimization

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). As a first step, you have to preform matching of the SIFT keypoints between previous frame and the current frame. 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. For example, at the start it will be the second frame. In that case from the previous frame, where you know the pose of the camera, you need to back-project SIFT features to the model by finding intersections with optical rays as in Exercise 1. Then those 3D points are projected into the current image with the pose from the previous frame and the reprojection error between those points and matched SIFT features between the frames has to be minimized. In Eq. 1 3D points $\mathbf{M_{i,t}}$ are obtained by back-projection as in Exercise 1.

As a final result, you have to compute and save: 1) the trajectory of the camera the world coordinate system, and 2) per-frame visualization of predicted poses as projection of 3D bounding box onto the image plane (as in Exercise 2).

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 $\mathbf{m}_{i,t}$ 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.

a) Implement the energy function **f** 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`
  Besides the lecture notes, for understanding the derivatives of exponential maps, we recommend the following resources:
  (i) *Unified Pipeline for 3D Reconstruction from RGB-D Images using Coloured Truncated Signed Distance Fields*[1], Section 3.3.1, Miroslava Slavcheva
  (ii) *Pose estimation for augmented reality: a hands-on survey*, Eric Marchand, Hideaki Uchiyama and Fabien Spindler
  (iii) *Odometry from RGB-D Cameras for Autonomous Quadrocopters*[2], Section 2.4, 2.5, 4.2 and 4.3, Christian Kerl
  (iv) *Practical Parameterization of Rotations Using the Exponential Map*, F. Sebastian Grassia

b) To estimate the camera pose you are asked to implement a Gauss-Newton or Levenberg Marquardt solver using the robust functions as given in this exercise. The solver should be implemented by yourself. We tried to provide details below, but more elaborate discussions take place in :

- *Multiple View Geometry*, Zissermann & Hartley
- *Robust Parameter Estimation in Computer Vision*, Charles V. Stewart
- *Pose estimation for augmented reality: a hands-on survey*, Eric Marchand, Hideaki Uchiyama and Fabien Spindler

A new formula for the derivatives of rotation matrix First, in the lecture as well as in the traditional aforementioned literature, we have given various formulas for the derivative of rotation matrix $\mathbf{R}$ w.r.t. its exponential coordinates.

In 2014, Guillermo Gallego and Anthony Yezzi have given a more compact, and maybe easier to grasp form in their paper:
A compact formula for the derivative of a 3-D rotation in exponential coordinates

[1] `http://campar.in.tum.de/personal/slavcheva/slavcheva_master_thesis.pdf`
[2] `https://vision.in.tum.de/_media/spezial/bib/kerl2012msc.pdf`

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 $\mathbf{v} \in \mathbb{R}^3$. Skew symmetric twist form $[\mathbf{v}]_x$ is related to $\mathbf{v}$ as:

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

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

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

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.

The choice of the derivative formulation is up to the implementation and you are free to choose any one of those as long as exponential maps are used. For double checking the Jacobians, 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.

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.

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:

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

Note that it is also valid to write separate error per u and per v coordinate. In that case your energy function is a two dimensional vector $\mathbf{E}_{2\times 1} = [E_u \ E_v]^T$ and your residuals are $N \times 1$ vectors $\mathbf{e_u}$ and $\mathbf{e_v}$ and there are separate Jacobians per coordinate error which are of dimensions $N \times 6$.

Algorithm 1 Levenberg Marquardt.

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

```
θ ← θ_0
t ← 0
λ ← 0.001
u ← τ + 1
for t < N and u > τ do
    J ← compute jacobian
    e ← E(x, θ)
    Δ ← −(J^T J + λI)^{-1}(J^T e)                ▷ compute update
    e_new ← E(x, θ + Δ)
    if e_new > e then
        λ ← 10λ
    else
        λ ← λ/10
        θ ← θ + Δ
    end if
    u ← ‖Δ‖
    t ← t + 1
end for
```

Levenberg Marquardt follows a similar path with a damped iteration and its pseudocode is given below using composed residual.

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:

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

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

where $c = 4.685$ is used based on the assumption of variance-1 with 95% rate in the outlier rejection. $\rho(e)$ denotes the actual Tukey loss, whereas $w(e_i)$ are the derivatives per residual component $\mathbf{e} = [e_0\ e_1\ \ldots\ e_N]^T$- which directly contribute to the gradient minimization:

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

where MAD is the median absolute deviations: $MAD(\mathbf{e}) = median(|\mathbf{e}|)$. $w(e_i)$ is used to fill the weight matrix $W_{2N\times 2N}$ used in the LM step.

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.

Algorithm 3 IRLS: Iteratively re-weighted least squares.

Require: Data $\{\boldsymbol{x}\}$, Initial parameters $\boldsymbol{\theta}_0$, Iterations N, 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 $t < N$ and $u > \tau$ **do**
  $\mathbf{e} \leftarrow [d_u(\mathbf{x}, \boldsymbol{\theta})\ d_v(\mathbf{x}, \boldsymbol{\theta})]^T$
  $\sigma \leftarrow 1.48257968\ \text{mad}(\mathbf{e})$ ▷ compute scale.
  $\mathbf{W}_{2N \times 2N} \leftarrow diag[\ldots, w_i(\mathbf{e}/\sigma; c), \ldots]$
  $e \leftarrow E(\mathbf{x}, \boldsymbol{\theta}) = \sum_i \rho(\mathbf{e}_i)$
  $\mathbf{J} \leftarrow \mathbf{J}(\mathbf{e})$
  $\Delta \leftarrow -(\mathbf{J}^T\mathbf{W}\mathbf{J} + \lambda\mathbf{I})^{-1}(\mathbf{J}^T\mathbf{W}\mathbf{e})$ ▷ compute update
  $e_{new} \leftarrow E(\mathbf{x}, \boldsymbol{\theta} + \Delta)$
  if $e_{new} > e_w$ **then**
    $\lambda \leftarrow 10\lambda$
  else
    $\lambda \leftarrow \lambda/10$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta$
  end if
  $u \leftarrow \|\Delta\|$
  $t \leftarrow t + 1$
end for