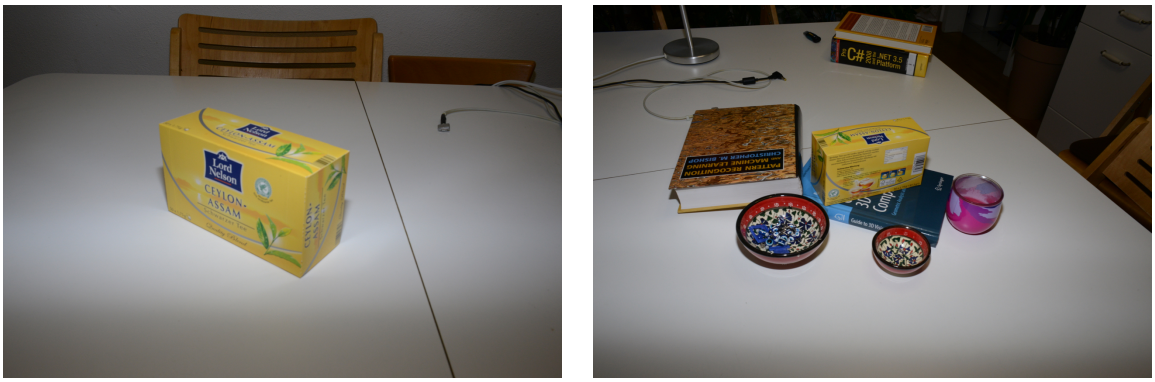
Technische Universität München
Fakultät für Informatik
PD Dr. Slobodan Ilic

WS 17/18
Exercise Sheet 3
November 2, 2017

Exercises in Tracking & Detection

In 3D Computer Vision object pose estimation is one of the key elements for model based tracking. Its applications are evident in many application areas like Augmented Reality, Robotics, Machine Vision and Medical Imaging. In this project you will develop a method that will estimate the pose of a camera in respect to the 3D model of the object exploiting the texture information of the object described by its visual features. You are given a calibrated camera with its known intrinsic parameters, 3D CAD model of the object described as a triangular mesh containing vertices and faces and a number of input RGB camera images. Initially the object doesn't contain associated texture, but you will be given couple of images of the object seen from different view points , which will serve to associate texture information to the object. The test images contain two sets. The first one are images depicting the object from different viewpoint and they are not consecutively taken, so have to be used separately for pose estimation. The second one is a video sequence and will be used for tracking the object. The project will contain several parts explained below and they should be implemented one after the other. The implementation has to be done in MATLAB. Please follow the requirements below in terms of the functions that can be used from MATLAB Toolboxes. If not specified that available functions from MATLAB can be used it means that the functions must be implemented by yourself.



(a) Teabox image for texturing. (b) Cluttered scene for detection.

Figure 1: Example images of the teabox for initial textring (a) and detection and pose refinement (b)

Task 1 Model preparation

As a first step, you will need to associate the texture information to the given 3D model from couple of input images depicting the object from different viewpoints. The 3D model is called *teabox.ply* given as ASCII file in PLY format. The description of the format can be found `http://paulbourke.net/dataformats/ply/`. The model contains 8 vertices accompanied with per vertex normals and 12 triangles (faces). The model can be visualized using Meshlab program `http://www.meshlab.net/`. In MATLAB use function **read_ply** provided on the course website.

In addition to the *teabox.ply* file describing the object's geometry you are also given the intrinsic camera parameters being: $f_x = f_y = 2960.37845$ $c_x = 1841.68855$ $c_y = 1235.23369$.

The world coordinate system is attached to lower left corner (vertex with coordinates [0 0 0]) of the object and it is right handed.

Your first task is to align the object to the input images that can be found in the folder **images\init_texture**. An example image is shown in Fig. 1a. The alignment can be made by clicking on the object vertices in the images providing 2D correspondences to the available 3D vertices of the object. The manual 3D-2D correspondences should be used with the PnP algorithm to compute the initial camera poses allowing alignment of the 3D model to the input texture images. The next step is to associate keypoints detected in the input texture images to the actual 3D model, thus creating additional 2D-3D correspondences that will be used in the following steps of the method. For detection of the keypoints, SIFT has to be extracted using VLFeat library `http://www.vlfeat.org/`.

Without loss of generality, a 2D point (x, y), map to a 3D ray using the inverse camera matrix:

$$\mathbf{r}(\mathbf{x}) = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \mathbf{K}^{-1} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

This is of course only true in the canonical configuration where the object/camera pose is assumed to be identity. When this is not the case one could always transform the ray/object to the correct frame of reference. Via normalization one should make sure that $\|\mathbf{r}(\mathbf{x})\| = 1$ is of unit length, as it depicts a direction.

Task 2 **Pose estimation with PnP**

Having the 3D model with associated 2D-3D correspondence and the camera intrinsics you have to estimate the camera pose in all subsequent test images given in the folder **images\detection** separately for each image. An example image is depicted in Fig. 1b. For that you need to use PnP and RANSAC. For PnP you can use *estimateWorldCameraPose* function from MATLAB's Computer Vision System Toolbox and RANSAC has to be implemented by yourself. The result of this procedure has to be the best pose hypothesis with the maximal number of inliers. You have to report in which images you manged to get close to correct pose and explain why.

The course webpage contains routines for ray-triangle intersection and mesh loading. The cameras poses when computed correctly appear as below:

Note that the texture in this figure is illustrative and you are not required to compute it.

Task 3 **Pose refinement with non-linear optimization**

After getting the best pose hypothesis from the previous step you need to do pose refinement. This includes minimizing the re-projection error between 3D points on the model corresponding to the detected feature points in the input texture images and the feature points detected in the image. This will result in writing an objective function that should be minimized in terms of the camera pose (rotation and translation).

We are now given an initial pose $[\mathbf{R}_0, \mathbf{T}_0]$, the intrinsic matrix $\mathbf{A}$ and the filtered 2D point correspondence pairs between $\mathcal{I}_0$ and $\mathcal{I}_{t=1\ldots n}$. Additionally, we have the 3D coordinates $\mathbf{M}_{i,0}$ of the 2D feature points in $\mathcal{I}_0$. 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:

$$\mathbf{f}_t(\mathbf{R}_t, \mathbf{T}_t; \mathbf{A}, \mathbf{M}_{i,0}, \mathbf{m}_{i,t}) = \sum_i \|\mathbf{A}\left(\mathbf{R}_t\mathbf{M}_{i,0} + \mathbf{T}_t\right) - \tilde{\mathbf{m}}_{i,t}\|^2 \tag{1}$$

Note that this objective function is defined for each test image $t = 1, \ldots, n$ from

images\detection.

a) 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.

b) Implement the energy function $\mathbf{f}$ in `MATLAB` that takes as input the rotation parameters (given in Exponential Maps), 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:

- While being taken as known input parameters: $\mathbf{f}$, $\mathbf{A}$ and $\mathbf{M}_i, \mathbf{m}_i$, 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.

- 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. You are required to use analytical Jacobians and finite-difference approximations should only be used for debugging purposes.
  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

c) We encourage you to implement a Gauss-Newton or Levenberg Marquardt solver using the robust functions as given in the exercise. The solver should be implemented by yourself but you could get help from MATLAB's **fminsearch** for debugging. We tried to provide details below, but more elaborate discussions take place in :

- *Multiple View Geometry*, Zissermann & Hartley

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

- *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
`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. Although, thanks to RANSAC, the initial pose is close to the desired one, it doesn't give a guarantee of the correct correspondences. Therefore, in the refinement stage, one should take explicit 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.

First of all, let us remind you that RANSAC inliers are already a good enough initialization and they should be used hereafter. 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 $N \times 6$.

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

Algorithm 1 Levenberg Mardquardt.

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
```

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:

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

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

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.

Algorithm 3 IRLS: Iteratively re-weighted least squares.

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

```
θ ← θ_0
t ← 0
λ ← 0.001
u ← τ + 1
for t < N and u > τ do
    e ← [d_u(x, θ) d_v(x, θ)]^T
    σ ← 1.48257968 mad(e)                          ▷ compute scale.
    W_{2N×2N} ← w(e/σ; c)
    e ← E(x, θ) = ρ(e^T e)
    J ← J(e)
    Δ ← −(J^T W J + λI)^{−1}(J^T W e)              ▷ compute update
    e_new ← E(x, θ + Δ)
    if e_new > e_w then
        λ ← 10λ
    else
        λ ← λ/10
        θ ← θ + Δ
    end if
    u ← ‖Δ‖
    t ← t + 1
end for
```

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.

Task 4 Tracking

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.

a) Estimate the current pose $[\mathbf{R}_t, \mathbf{T}_t]$ at time t by using the 3D-2D correspondences $\mathbf{M}_{i,0}, \mathbf{m}_{i,t}$ and the pose parameter at time $(t-1)$ as initial values. This is done by minimizing $\mathbf{f}_t$ in Eq. 1, which is a non-linear least squares problem.

b) Repeat Exercise 3b for all images $\mathcal{I}_{t=1\ldots n}$ (namely, construct and solve all $\mathbf{f}_{t=1\ldots n}$). Show the trajectory of the camera in a 3D graph by plotting the camera coordinates in the world coordinate frame. The camera locations in the world coordinate frame at time t are obtained by computing $-\mathbf{R}_t^\top\mathbf{T}_t$. Again, save all these results in advance to avoid re-computing them during the correction.