Technische Universität München
Fakultät für Informatik
PD Dr. Slobodan Ilic
Shugurov, Tomczak, Haarbach,
Hamad, Fu, Yu

WS 20/21
Project 1
November 5, 2020

Tracking & Detection in Computer Vision

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, a 3D mesh model of the object and finally a number of input RGB camera images. Initially the object doesn't contain associated texture, but you will be given a couple of images of the object seen from different viewpoints, which will serve to associate texture information to the object.

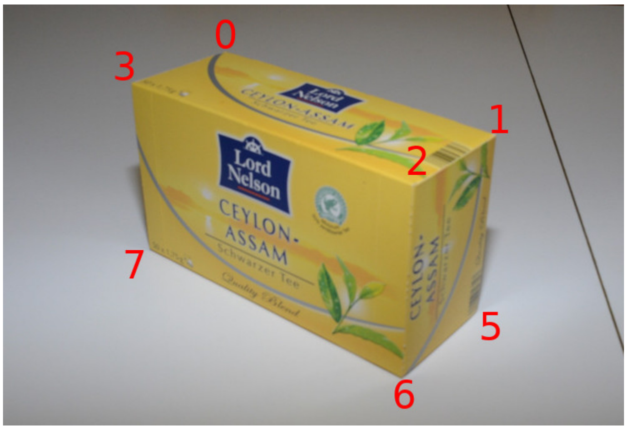


Figure 1: Teabox model with its 8 vertices (vertex 4 is hidden).

Task 1 Model preparation and SIFT keypoint extraction

As a first step, you will need to associate the texture information to the given 3D model from a couple of input images depicting the object from different viewpoints. The 3D model called **teabox.ply** is given as an ASCII[1] file in PLY format.[2] It consists of 8 vertices defining the corners of the box accompanied with per vertex normals and 12 triangular faces.[3] The origin of the right handed world coordinate system coincides with the lower left corner of the object in Figure 1 which is vertex 7 (0-based indexing) in the .ply file. In addition to the PLY file describing the object's geometry you are also given the intrinsic camera parameters: $f_x = f_y = 2960.37845$ $c_x = 1841.68855$ $c_y = 1235.23369$.

a) For each image in the folder **init_texture** we would like to know the pose of the object. We can manually extract the 2D locations of the object's visible corners in each image.[4] Then, given 2D coordinates of the corners in the image and the corresponding 3D coordinates of the vertices in the model, the object's pose can be estimated using PnP. For PnP you can use `solvePnP` from OpenCV's `calib3d` module.

[1] so it can be inspected with any text editor.

[2] The description of the .ply format can be found at `http://paulbourke.net/dataformats/ply/`

[3] As can be seen by turning on wireframe rendering in Meshlab `http://www.meshlab.net/`

[4] e.g. using paint/gimp or integrated via clicking with the mouse using OpenCV's `MouseCallback`

Expected outcome The computed camera poses (rotation and translation) have to be shown, e.g. as in Figure 2. Note that we do not require to project texture onto the object.

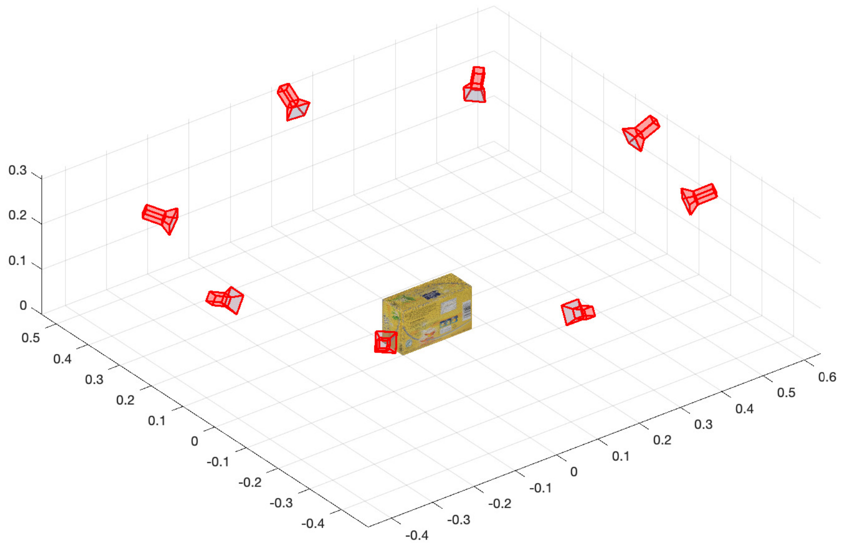


Figure 2: Camera poses

b) The next step is to detect SIFT keypoints in the images. SIFT keypoints can be extracted using `SIFT::create()` and `detectAndCompute()` from OpenCV's `features2d` module. However, we only want to keep those keypoints that arise from the models texture, not from the background. Since we know the model as well as the camera pose for each image we can filter them as follows.

Without loss of generality, a 2D point $\mathbf{m} = [x, y]^T$, maps to a 3D ray using the inverse intrinsics matrix and the camera optical center C:

$$\mathbf{r}(\mathbf{m}) = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \mathbf{C} + \lambda \mathbf{Q}^{-1}\mathbf{m}$$

where $\mathbf{Q}$ is part of the projection matrix $\mathbf{P} = [\mathbf{Q}\ \mathbf{q}]_{3\times 4} = \mathbf{K}[\mathbf{R}|\mathbf{t}] = [\mathbf{KR} \mid \mathbf{Kt}]$ and $\mathbf{C} = -\mathbf{Q}^{-1}\mathbf{q}$ is the optical center of the camera.

Given the ray $\mathbf{r}(\mathbf{m})$, you need to find out where it intersects the model in 3D space. Since the teabox happens to be aligned with the axis of its model coordinate system it coincides with its Axis-aligned minimum bounding box (AABB). We can thus just extract its extent from the 8 vertices in the .ply file, ignoring its triangle faces, and use a simple Ray-Box intersection method[5] that has to be implemented.

This allows to back-project the 2D SIFT features onto the 3D model. These 2D-3D correspondences will be used for automatic object detection and pose estimation (no more mouse clicking) in the following exercises, so for each SIFT keypoint you need to save its descriptor and 3D location in the model coordinate system.

Expected Outcome In order to verify that backprojection has been implemented correctly, you need to visualize the locations of the SIFT keypoints on the 3D model.

[5] `https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shape`
`ray-box-intersection`

Task 2 Pose estimation with PnP and RANSAC

After the last exercise we have SIFT keypoints corresponding to the texture of teabox and their 3D locations on the model. Now object detection and pose estimation can be automatized.

For each image in the folder **detection** you are required to detect the object and estimate its pose. Given an image, you need to compute SIFT keypoints and match them to the database of SIFT keypoints computed in the previous exercise. For matching of the keypoints you should use `DescriptorMatcher::match` . Those matches will provide 2D-3D correspondences, which can be used for PnP to estimate object's pose.

Unfortunately, in contrast to the last task, this time PnP cannot be applied directly since there are some wrong matches which will worsen the estimated pose. This problem can be overcome by using the RANSAC algorithm. The following high-level pseudocode is based on RANSAC description in "Multiple View Geometry" by Richard Hartley and Andrew Zissermann.

RANSAC

i Randomly select a sample of **4** data points from S and estimate the pose using PnP.

ii Determine the set of data points S_i from all 2D-3D correspondences where reprojection error (Euclidean distance) is below the threshold t. The set S_i is the consensus set of the sample and defines the inliers of S.

iii If the number of inliers is greater than we have seen so far, re-estimate the pose using S_i and store it with the corresponding number of inliers.

iv Repeat the above mentioned procedure for N iterations.

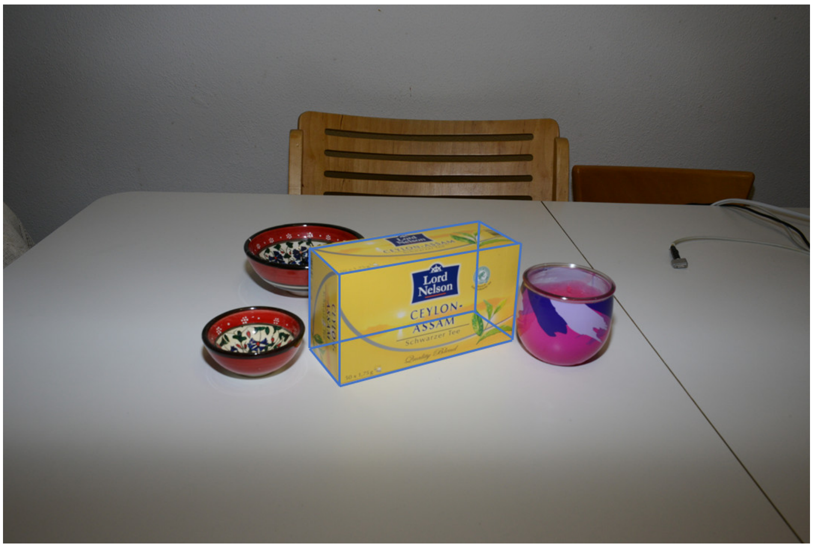


Figure 3: Visualization of the bounding box for the estimated pose

There is no need to implement this algorithm from scratch, just use `solvePnPRansac()` and experiment with different values for threshold t and number of iterations N.

Expected Outcome For each image you are required to provide a visualization of the projected 3D bounding box of the detected object as shown in Figure 3.

Task 3 Pose refinement with non-linear optimization

Problem description The final task is to preform tracking of the camera with respect to the given 3D model.

As initialization, detect the object in the first image I_0 in the folder **tracking** 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 task 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 pixel location of the 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 pairs of matching 2D image points $\tilde{\mathbf{m}}_{i,t}$ and 3D coordinates $\mathbf{M}_{i,t}$. In order to compute the current camera pose $[\mathbf{R}_t|\mathbf{t}_t]$, we formulate an energy or objective function $\mathbf{f}_t$ and apply non-linear optimization tools. One possible formulation of $\mathbf{f}_t$ is the sum of 2D reprojection errors from all point correspondences as follows:

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

The camera matrix used in (back-)projection is composed from the intrinsic parameters given in Task 1 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 distortion effect for the lens. While this is not true in practice, and will result in slightly perturbed correspondences, it should be a good approximation for the purposes of this exercise.

Implementation tasks You are required to implement an Iterative Reweighed Least Square(IRLS) method for non-linear optimization of the camera pose parameters in the provided tracking sequence. This method is a robust version of the Levenberg-Marquart (LM) optimization algorithm which itself is a mixture of the Gauss-Newton method (GN) and simple gradient descent. IRLS can handle a certain amount of outliers using a robust estimator loss. The pseudo code for IRLS algorithm is given below as Algorithm 1. However, you are not required to implement this algorithm from scratch, since we will be using Ceres Solver `http://ceres-solver.org/` as a basis abstracting away the tricky parts.

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 $\tau$
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
      $\mathbf{e}_{2N\times 1} \leftarrow [d_u(\mathbf{x}, \boldsymbol{\theta})\ d_v(\mathbf{x}, \boldsymbol{\theta})]^T$          ▷ $\mathbf{e} = [e_1\ e_2\ \dots\ e_{2N}]^T$
      $\sigma \leftarrow 1.48257968\ \mathrm{MAD}(\mathbf{e})$          ▷ compute scale.
      $\mathbf{W}_{2N\times 2N} \leftarrow diag[\dots, w(e_i/\sigma), \dots]$
      $E(\mathbf{x}, \boldsymbol{\theta}) = \sum_i^{2N} \rho(e_i/\sigma)$
      $\mathbf{J} \leftarrow \mathbf{J}(\mathbf{e})$
      $\Delta \leftarrow -(\mathbf{J}^T\mathbf{W}\mathbf{J} + \lambda\mathbf{I})^{-1}(\mathbf{J}^T\mathbf{W}\mathbf{e})$          ▷ 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 read up the official Ceres tutorial in order to understand what actually needs to be implemented and what is already provided for by the framework. In particular, you need to consider the following three points to model IRLS for re-projection error minimization in a general non-linear least squares solver such as Ceres:

a) **The Energy Function** (denoted with $\mathbf{f}$ in eq. 1 or $E(x, \boldsymbol{\theta})$ in Algo. 1) that takes as input the camera pose $[\mathbf{R}|\mathbf{t}]$, suitably parameterized via the parameter vector $\boldsymbol{\theta}$, the intrinsic matrix $\mathbf{A}$ and all the 3D-2D correspondences $\mathbf{M}_i, \mathbf{m}_i$ is called objective function in Ceres. For each of its terms, a `CostFunction` is responsible for computing a vector of residuals and Jacobian matrices.

   - $\mathbf{A}$ and $\mathbf{M}_i, \mathbf{m}_i$ are known, the only unknown quantity that has to be optimized for is the parameter vector $\boldsymbol{\theta}$ representing the camera pose $[\mathbf{R}|\mathbf{t}]$ consisting of rotation and translation.
   - After projecting 3D points to the 2D image plane, one has to convert from homogeneous coordinates to Cartesian coordinates by dividing 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 the parameters of its representation as a 9-element rotation matrix $\mathbf{R} \in SO(3)$ while enforcing the constraints that the special orthogonal group implies is infeasible for an optimization algorithm. For this, it is better to use a more compact local parameterization such as the exponential map (also known as Angle-Axis) or unit quaternions. For conversion, you may use Ceres rotation.h[6] or the Geometry module [7] of the Eigen library.

[6] `https://github.com/ceres-solver/ceres-solver/blob/master/include/ceres/rotation.h`

[7] `https://eigen.tuxfamily.org/dox/group__Geometry__Module.html`

b) **Jacobian computation.** The second step is to find the derivatives of the objective function and to create its Jacobian matrix. Since this is an errorneous task if done by hand, we follow `http://ceres-solver.org/derivatives.html` to advice:

(i) Use automatic differentiation with `AutoDiffCostFunction` . For this all the computations needed inside the energy function must be implemented in a templated way. This can be done by hand operating directly on scalar or using Ceres rotation.h or Eigen Geometry module.

(ii) If this is not possible (e.g. if a function from an external library such as OpenCV was used to project the points), use numerical derivates computed via finite differences. This is slow but serves also to check to correctnes of the analytic derivates in the next step. In Ceres, for this use case, there is actually no need to implement anything. Just set the `Solver::Options::check_gradients` and the framework computes and compares the gradients for you.

(iii) Try to implement analytic derivates correctly. This is hard to get right, but if done correctly ususally results in a good performance. The chain rule needs to be used in order to compute the derivate of the 2D reprojection error x, y wrt. to the parameterization θ. One important aspect is the choice of parameterization and derivative formulation for the rotational part which 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.

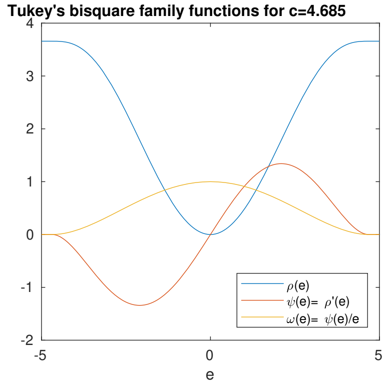
- In Ceres, this concept of Manifold Optimization can generally be modelled via a `LocalParameterization` of the rotational part that is used in the update step which naturally constrains the optimization to stay on the manifold and improves the numerical behaviour of the algorithm. You may experiment with `QuaternionParameterization` or the `EigenQuaternionParameterization` with differnt memory layout or with `Sophus::SO3` .

c) **Outlier treatment** Due to wrong SIFT correspondences the input might be corrupted by outliers. Therefore, one should take particular care of the outliers. While there are many ways of doing that, we will be using robust norms, also called M-estimators.

In Ceres, outlier treatment for robust curve fitting[8] in the end just boils down to providing a `LossFunction` and the right parameter to separate inliers from outliers.

[8] `http://ceres-solver.org/nnls_tutorial.html#robust-curve-fitting`

There are many robust loss functions (such as Cauchy or Huber) that could be used, but we focus on Tukey's bisquare (or biweight) family of functions as an M-estimator because the Tukey loss function $\rho(e)$ (3), (*In Ceres* `TukeyLoss`) assigns constant weight to outliers. Using Tukey (and different to Huber, Cauchy) outliers do not influence the optimization at all since the magnitude of their gradient $\psi(e)$ (4) is set to zero via the weighting function $w(e)$ (5). We use the tuning constant $c = 4.685$ which is based on the assumption of unit variance with 95% rate in outlier rejection.



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

$$\psi(e) = \begin{cases} e\left(1-\left(\frac{e}{c}\right)^2\right)^2 & \text{if } |e| < c \\ 0 & otherwise \end{cases} = \rho'(e) \tag{4}$$

$$w(e) = \begin{cases} \left(1-\left(\frac{e}{c}\right)^2\right)^2 & \text{if } |e| < c \\ 0 & otherwise \end{cases} = \psi(e)/e \tag{5}$$

Expected outcome We expect you to do the following:

- Implement the IRLS algorithm, initialize and then run it on the provided sequence.
- Compute, save and visualize the trajectory of the camera as in Task 1.
- Compute, save and visualize the projection of the 3D bounding box onto the image plane for each frame as in Task 2.

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.