Computer Vision Assignment 01

Luca Ebner (ebnerl@student.ethz.ch)

October 1, 2020

1. Data Normalization (2.1)

In order to improve numerical stability, the stored 2D points $\mathbf{x_i} = [x_i, y_i]$ and 3D points $\mathbf{X_i} = [X_i, Y_i, Z_i]$ are normalized before computing the camera matrix $\mathbf{P}$. To achieve this, the following procedure is done:

1. 2x1 vectors of the n stored 2D points are extended by a one (homogenization)
2. Points centroid is shifted to the origin
3. Points are scaled to have unit mean distance to the origin

We are looking for the matrices $\mathbf{U}, \mathbf{T}$ such that $\hat{\mathbf{x}} = \mathbf{T}\mathbf{x}$ and $\hat{\mathbf{X}} = \mathbf{U}\mathbf{X}$, where $\hat{\mathbf{x}}$ and $\hat{\mathbf{X}}$ are the normalized matrices including the calibration point pairs. As it turns out, $\mathbf{U}, \mathbf{T}$ can be computed as follows:

$$\mathbf{T} = \begin{bmatrix} d_{xy} & 0 & c_x \\ 0 & d_{xy} & c_y \\ 0 & 0 & 1 \end{bmatrix}^{-1}, \mathbf{U} = \begin{bmatrix} d_{xyz} & 0 & 0 & c_X \\ 0 & d_{xyz} & 0 & c_Y \\ 0 & 0 & d_{xyz} & c_Z \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1},$$

where d_{xy}, d_{xyz} is the mean distance to the origin and c_{xy}, c_{XYZ} is the centroid.

2. Direct Linear Transform (2.2)

To estimate the normalized camera matrix $\hat{\mathbf{P}}$, the normalized 2D and 3D point correspondences are used. To successfully compute $\hat{\mathbf{P}}$, a minimum of 6 points is required. Each point pair is inserted into this equation:

$$\begin{bmatrix} \hat{\mathbf{X}}_i^\top & \mathbf{0}_{1\times4} & -\hat{x}_i\hat{\mathbf{X}}_i^\top \\ \mathbf{0}_{1\times4} & -\hat{\mathbf{X}}_i^\top & \hat{y}_i\hat{\mathbf{X}}_i^\top \end{bmatrix} \begin{pmatrix} \hat{\mathbf{P}}^1 \\ \hat{\mathbf{P}}^2 \\ \hat{\mathbf{P}}^3 \end{pmatrix} = \mathbf{0}$$

If we stack the equations of all point correspondences, we can then compute $\hat{\mathbf{P}}$.

After the computation of $\hat{\mathbf{P}}$, it is transformed back by computing $\mathbf{P} = \mathbf{T}^{-1}\hat{\mathbf{P}}\mathbf{U}$. With $\mathbf{P}$, we can reproject the 3D points back into 2D space and compare them with the points we clicked. If done correctly, the reprojected and the original points should look close to identical.

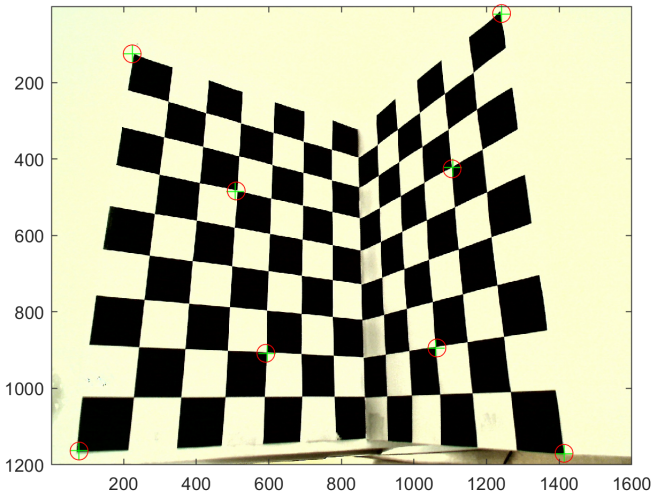

Figure 1: Reprojection (red circles) compared to clicked points (green cross)

***Factorize the camera matrix into the intrinsic camera matrix* $\mathbf{K}$*, a rotation matrix* $\mathbf{R}$ *and the camera center* $\mathbf{C}$**

The underlying equation here is $\mathbf{P} = \mathbf{K}[\mathbf{R}| - \mathbf{RC}]$. After factorizing the camera matrix I tested the decomposition by putting $\mathbf{P}$ back together from $\mathbf{K}$, $\mathbf{R}$ and $\mathbf{t} = -\mathbf{RC}$:

```
% === Task 2 DLT algorithm ===
[P, K, R, t, error] = runDLT(xy, XYZ)
visualization_reprojected_points(xy, XYZ, P, IMG_NAME);

%Test if decomposition is correct:
visualization_reprojected_points(xy, XYZ, K*[R t], IMG_NAME);
```

Indeed, the two visualized figures look identical. The decomposition therefore is correct.

Visualize the reprojected points of all checkerboard corners on the calibration object with the computed camera matrix.

To get a matrix with all the 3D checkerboard corners, I used the code seen below. I then visualized these points on the checkerboard image seen in Figure 2.

```
for i=0:x_dim
    for j=0:z_dim
        corner = [i*spacing; 0; j*spacing];
        XYZ_corners = [XYZ_corners, corner]
    end
end
for j=0:z_dim
    for k=1:y_dim
        corner = [0; k*spacing; j*spacing];
        XYZ_corners = [XYZ_corners, corner];
    end
end

```

```
% Apply projection matrix P
XYZ_homogeneous=homogenization(XYZ);
xyz_projected=P*XYZ_homogeneous;
```

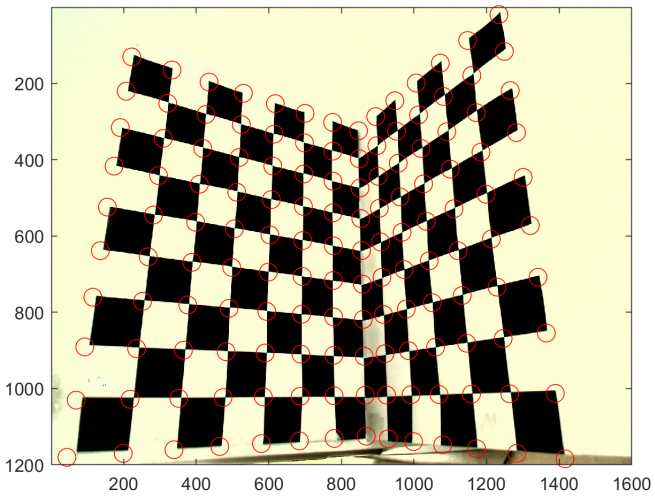

Figure 2: Checkerboard corners visualization

It can be seen that points closer to the center of the image are more precise, whereas points towards the border are slightly off. To cope with this, one could use more than 6 calibration points or points that are closer to the border, that way one can obtain a better estimate of $\mathbf{P}$.

What happens if you use the unnormalized points?

If we were to compute $\mathbf{P}$ directly, meaning without the normalization and backtransformation, the computation might become unstable. To test this, I inserted the following code inside `runDLT.m`:

```
%compute P with normalized calibration points
[Pn] = dlt(xy_normalized, XYZ_normalized);
[P] = inv(T)*Pn*U

%compute P without normalized calibration points
[P] = dlt( [xy; ones(1,size(xy,2))], [XYZ; ones(1,size(xy,2))] )
```

On a first look, the resulting $\mathbf{P}$ was very different. Interestingly, the reprojected points visualized in the figures still seemed to be the same. After a closer inspection I found that the difference between the two $\mathbf{P}$'s was mostly in scale, thus leading to almost the same points. However, it is important to note that apart from the scale factor there still was a difference! I can imagine that, if you use points with magnitude of different orders, the two different computations of $\mathbf{P}$ could differ even more.

3. Gold Standard Algorithm (2.3)

From the reprojection of all checkerboard corners described in the previous section we have seen that the estimation of $\mathbf{P}$ is not perfect. One of the possible reasons is that the 2D points that we clicked and the

corresponding 3D point that we entered manually do not correspond *exactly*, we can look at this problem like a measurement error.
If we reproject the 3D points back into 2D space, we can observe that they are not exactly at the same spot as their originals. Therefore, we want to optimize $\mathbf{P}$ to minimize the geometric error between the original and the reprojected 2D points to get the best fit.

To do this, we define a cost function to penalize deviations of the reprojected points to their originals and optimize $\mathbf{P}$ accordingly. If we then visualize the checkerboard corners again and zoom in on one of the corners, we can see that the accuracy is improved.

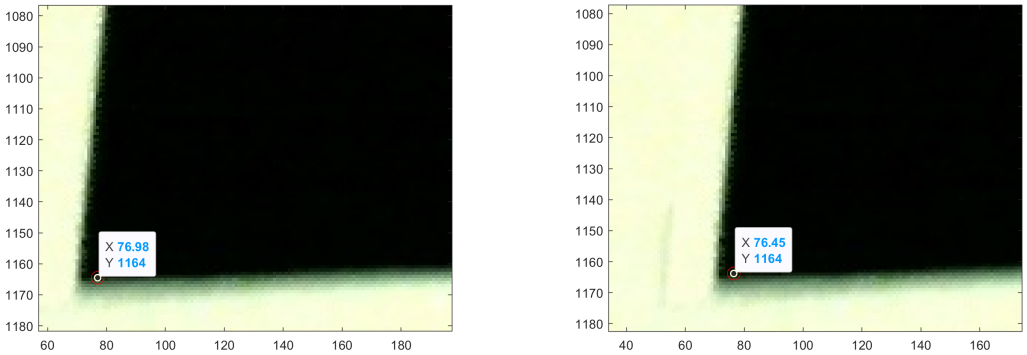

Figure 3: Zoomed in on bottom left corner before (left) and after (right) optimization