Laboratory Session 4: Bundle Adjustment and Multiview Geometry

In this laboratory session we will implement the complete pipeline for pose estimation between three cameras and a sparse 3D reconstruction of points of the scene using multi-view geometry for a rough estimation and a Bundle Adjustment for a refinement.

Goals of the assignment:

1. Multiview Geometry. Understanding the problem of recovering the relative pose up to scale between more than two views.
2. Understanding Bundle Adjustment and how to define a residual function.

Evaluation of the assignment:

The resulting work will be shown by presenting the obtained results in the following Laboratory session, and the code will be submitted through the ADD (Moodle).

1. Line fitting example using least-squares

We provide the file `lineFitting.py` where a 2D line is fitted using least-squares function. Least-squares minimize a loss with is the quadratic error of a set of distances (typically called residuals). Each residual can be also understood as a homogenous equation that ideally should be zero.

```
# Optimization with L2 norm and Levenberg-Marquardt
OpOptim = scOptim.least_squares(resLineFitting, Op, args=(xData,), method='lm')
```

This function compute a loss using the residuals array (such that `loss = res.T @ res`) and minimize this loss in order to fit the model.
Notice that you need a function computing the residuals such that, given an array `Op` containing the variables of the model to optimize, and a set of additional fixed parameters (i.e. the data to fit and calibration parameters), it returns an array of residuals (at least as crowded as the number of variables to optimize).

```
def resLineFitting(Op,xData):
    """
    Residual function for least squares method
    -input:
      Op: vector containing the model parameters to optimize (in this case the
line). This description should be minimal
      xData: the measurements of our model whose residual we want to calculate
    -output:
      res: vector of residuals to compute the loss
    """
```

In general it is a good idea that the number of parameters to optimize corresponds to the degrees of freedom of the geometric element to fit (e.g. two parameters for a line, three for a plane, etc…). If not, we must impose additional constraints.

2. Pose estimation and bundle adjustment

For this session, we provide a set of 35 perfectly matched points among three images, without spurious data. We are going to focus on the geometric aspects of the 3D reconstruction avoiding the problem of outliers' detection and robust fitting.
We also provide the complete ground truth of the camera references, 3D points and scale in order to you to identify bugs and to partition your algorithm.
In file `plotGroundTruth.py` you can find an example of plotting the ground truth data that you can use for plotting your own results.

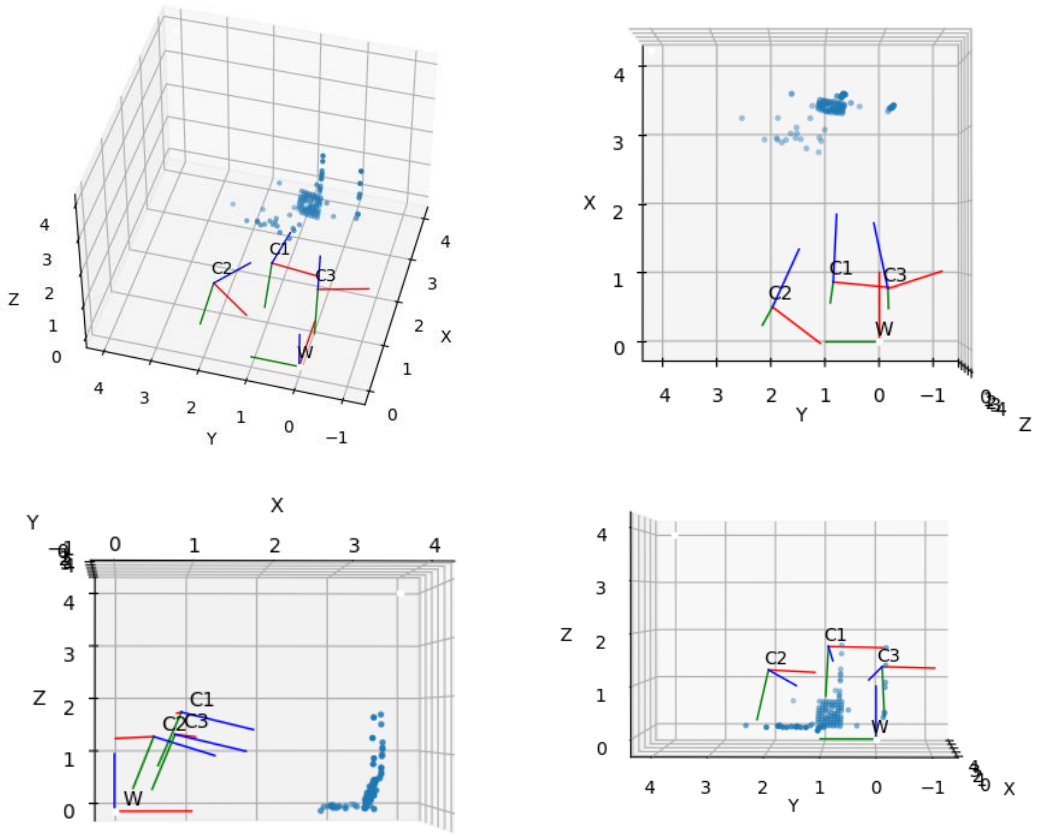

Fig.1 Example of 3D plot of the ground truth data.

The starting point of the practice is the result from laboratory session 2 (i.e. the estimated pose and the 3D points reconstructed from the two views) which corresponds to an initial linear solution obtained from a fundamental matrix. You can reuse your code from lab 2 and get a solution from the following fundamental matrix example (also included in the support files),

```
F_21 = [[ 0.00022244,   0.000624  ,   0.10418026],
        [ -0.00015211,  -0.00004897,   0.60030525],
        [ -0.30234655,  -0.71224975, 100.        ]]
```

or any of the ones obtained in previous labs.

2.1. Bundle adjustment from two views

Implement a residual function for the bundle adjustment problem from two views. Be careful with the cameras and points description in the optimization parameters and their conversion to the linear expressions. For rotations, we recommend to use the description of 3 parameters $\boldsymbol{\theta} \in so(3)$ (see Appendix A).

```
def resBundleProjection(Op, x1Data, x2Data, K_c, nPoints):
    """
    -input:
        Op: Optimization parameters: this must include a
paramtrization for T_21 (reference 1 seen from reference 2)
            in a proper way and for X1 (3D points in ref 1)
        x1Data: (3xnPoints) 2D points on image 2 (homogeneous
coordinates)
        x2Data: (3xnPoints) 2D points on image 2 (homogeneous
coordinates)
```

```
        K_c: (3x3) Intrinsic calibration matrix
        nPoints: Number of points
    -output:
        res: residuals from the error between the 2D matched points
and the projected points from the 3D points
             (2 equations/residuals per 2D point)
    """
```

Test the function with the initial solution obtained from the essential matrix. Check the projection of the points from this solution and visualize the residuals.

Implement a bundle adjustment using `least-squares` function.

Visualize the 3D results of your solution (cameras and points) and compare them with the provided ground truth.

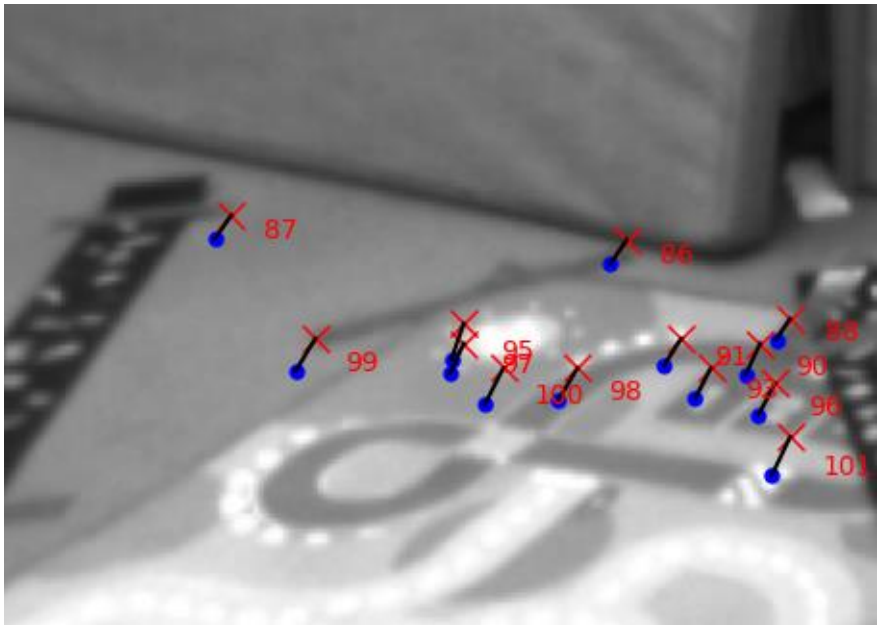

Fig.2. Residuals visualization: In red cross the 2D points from matching. In blue circle the projection of the 3D points using poses and calibration.

3. Perspective-N-Point pose estimation of camera three

From the 3D points triangulated in section 2 and the provided matched points compute the pose of the camera 3 with respect to the camera 1 using `cv2.solvePnP` function.

```
retval, rvec, tvec  = cv2.solvePnP(objectPoints, imagePoints, K_c,
distCoeffs,flags=cv2.SOLVEPNP_EPNP)
```

The way that this function deals with image points is quite confusing (and not properly documented). From an homogenous matrix `x` with size `3xn` consider defining the image points as,

```
imagePoints = np.ascontiguousarray(x[0:2,
:].T).reshape((x.shape[1], 1, 2))
```

Since we have previously undistorted the image the distortion coefficients must be zero. rvec is a compact representation of the rotation (see Appendix 1).

4. Bundle adjustment from 3 views

Modify your previous implementation of the residual function in order to implement a bundle adjustment for three or more views. Be careful with the cameras definition and the differences between the two first cameras transformation and the rest of the cameras transformations in terms of degrees of freedom.

Scale the obtained results with the scale of the ground truth transformation `T_12` and compare the results.

Appendix A

Consider the rotation of an angle θ around the axis defined by the unitary vector $\hat{\mathbf{\theta}}$ that defines the rotation from basis 1 to basis 2 (see Euler's rotation theorem).

The vector $\mathbf{\theta} = \begin{pmatrix} \theta_1 & \theta_2 & \theta_3 \end{pmatrix}^T$ where $\mathbf{\theta} = \theta\hat{\mathbf{\theta}}$ is 3 parameters representation of the rotation such that,

$$\mathbf{R} = \exp\left(\left[\mathbf{\theta}\right]_\times\right), \text{ which is called the exponential mapping where}$$

$$\left[\mathbf{\theta}\right]_\times = \begin{pmatrix} 0 & -\theta_3 & \theta_2 \\ \theta_3 & 0 & -\theta_1 \\ -\theta_2 & \theta_1 & 0 \end{pmatrix} \text{ is the anti-symmetric matrix of } \mathbf{\theta}.$$

The opposite operation is called the logarithmic mapping and can be computed as

$$\left[\mathbf{\theta}\right]_\times = \log\left(\mathbf{R}\right)$$

where exp and log represent the exponential and the logarithm of matrices.

```
def crossMatrixInv(M):
    x = [M[2, 1], M[0, 2], M[1, 0]]
    return x

def crossMatrix(x):
    M = np.array([[0, -x[2], x[1]],
                  [x[2], 0, -x[0]],
                  [-x[1], x[0], 0]], dtype="object")
    return M

R = np.expm(crossMatrix(theta))

theta = crossMatrixInv(np.logm(R))
```

WARNING!

When computing `scipy.linalg.logm(R)` it can happen that you get a vector with imaginary components and a warning message like: "`logm result may be inaccurate, approximate err = 3.8252542590646475e-07`".

This is caused by a lack of computational precision of working with float32. The problem is solved if you cast the input to float64, i.e. `scipy.linalg.logm(R.astype('float64'))`.