EE 046746 - Technion - Computer Vision

Homework 4 - Structure From Motion

Due Date: 26.01.2022

Submission Guidelines

READ THIS CAREFULLY

- Submission only in **pairs**.
- **No handwritten submissions**.
- You can choose your working environment:
  - You can work in a `Jupyter Notebook`, locally with Anaconda or online on Google Colab
    - Colab also supports running code on GPU, so if you don't have one, Colab is the way to go. To enable GPU on Colab, in the menu: `Runtime` → `Change Runtime Type` → `GPU`.
  - You can work in a Python IDE such as PyCharm or Visual Studio Code.
    - Both also allow opening/editing Jupyter Notebooks.
- You should submit two **separated** files:
  - A compressed `.zip` file, with the name: `ee046746_hw4_id1_id2.zip` which contains:
    - A folder named `code` with all the code files inside (`.py` or `.ipynb` ONLY!), and all the files required for the code to run.
      - **The code should run both on CPU and GPU without manual modifications**, require no special preparation and run on every computer.
    - A folder named `output` with all the output files that are not required to run the code. This includes visualizations and saved data that are not included in your PDF report.
  - A report file (visualizations, discussing the results and answering the questions) in a `.pdf` format, with the name `ee046746_hw4_id1_id2.pdf`.
    - Be precise, we expect on point answers. **But don't be afraid to explain you statements (actually, we expect you to).**
      - Even if the instructions says "Show...", you still need to explain what are you showing and what can be seen.
  - No other file-types (`.docx`, `.html`, ...) will be accepted.
- Submission on the course website (Moodle).

Python Libraries

- `numpy`
- `matplotlib`
- `opencv` (or `scikit-image`)
- `scikit-learn`
- `scipy`
- Anything else you need (`os`, `pandas`, `csv`, `json`,...)

In [ ]:

```
import cv2 as cv
```

```
import numpy as np
import scipy.optimize
import numpy.linalg as la
import matplotlib.pyplot as plt

# for visualization windows to pop out in jupyter (kernel may require restart after using GUIs)
%matplotlib qt
```

Tasks

- In all tasks, you should document your process and results in a report file (which will be saved as `.pdf`).
- You can reference your code in the report file, but no need for actual code in this file, the code is submitted in a seprate folder as explained above.

Introduction

One of the major areas of computer vision is 3D reconstruction. Given several 2D images of an environment, can we recover the 3D structure of the environment, as well as the position of the camera/robot? This has many uses in robotics and autonomous systems, as understanding the 3D structure of the environment is crucial to navigation. You don't want your robot constantly bumping into walls, or running over human beings!

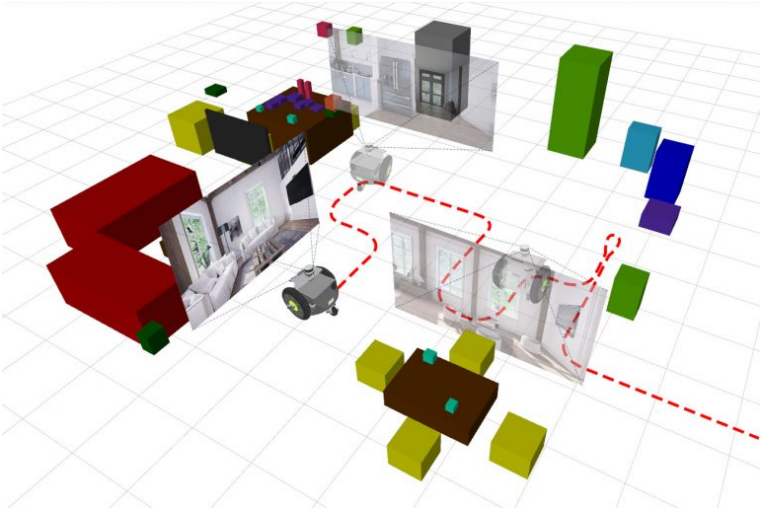


Image Source - CVPR 21 Embodied AI Workshop

In Part 1, you will be writing a set of functions to generate a sparse point cloud for some test images we have provided to you. The test images are 2 renderings of a temple from two different angles. We have also provided you with a `npz` file containing good point correspondences between the two images. You will first write a function that computes the fundamental matrix between the two images. Then write a function that uses the epipolar constraint to find more point matches between the two images. Finally, you will write a function that will triangulate the 3D points for each pair of 2D point correspondences.

In Part 2 (Bonus), *if you wish to do it*, you will be writing a set of functions to calibrate a camera and project a 3D CAD model to a 2D image after estimating the camera pose. We have provided you with a `npz` file containing corresponding 2D-3D pairs. You will first write a function that estimates a camera matrix given 2D-3D calibration points. Then write a function to decompose the estimated camera matrix to intrinsic/extrinsic parameters. Finally, you will write a script to project the provided 3D CAD model and compare it to a given 2D image of an airplane.

Part 1 - Sparse Reconstruction

In this section, you will be writing a set of function to compute the sparse reconstruction from two sample images of a temple. You will first estimate the Fundamental matrix, compute point correspondences, then plot the results in 3D. It may be helpful to read through Section 1.6 right now. In Section 1.6 we ask you to write a testing script that will run your whole pipeline. It will be easier to start that now and add to it as you complete each of the questions one after the other.

1.1 - Eight Point Algorithm

In this question, you're going to use the eight point algorithm which is covered in class to estimate the fundamental matrix. Please use the point correspondences provided in `data/some_corresp.npz`; you can load and view the contents of a `.npz` file as follows:

> `data = np.load("../data/some_corresp.npz")`
> print(data.files)

- Write the following function:

In [ ]:
```python
def eight_point(pts1, pts2, pmax):
    """
    Eight Point Algorithm
    [I] pts1, points in image 1 (Nx2 matrix)
        pts2, points in image 2 (Nx2 matrix)
        pmax, scalar value computed as max(H1,W1)
    [O] F, the fundamental matrix (3x3 matrix)
    """
    # replace pass by your implementation
    pass
```

where `pts1` and `pts2` are $N \times 2$ matrices corresponding to the (x, y) coordinates of the N points in the first and second image respectively, and `pmax` is a scale parameter. Implementation tips:

- Normalize points and un-normalize F: You should scale the data by dividing each coordinate by $p_{\max}$ (the maximum of the image's width and height) using a transformation matrix T. After computing F, you will have to "unscale" the fundamental matrix. If $p_{\text{norm}} = Tp$, then $F_{\text{unnorm}} = T^T F T$. Note that this scaling is slightly simpler than "centering" that you did in the lecture, but for the purpose of this assingment it should suffice.

- You must enforce the rank 2 constraint on F before unscaling. Recall that a valid fundamental matrix F will have all epipolar lines intersect at a certain point, meaning that there exists a non-trivial null space for F. In general, with real points, the eight-point solution for F will not come with this condition. To enforce the rank 2 constraint, decompose F with SVD to get the three matrices U, Σ, V such that $F = U\Sigma V^T$. Then force the matrix to be rank 2 by setting the smallest singular value in Σ to zero, giving you a new Σ'. Now compute the proper fundamental matrix with $F' = U\Sigma' V^T$.

- You may find it helpful to refine the solution by using local minimization. This probably won't fix a completely broken solution, but may make a good solution better by locally minimizing a geometric cost function. For this we have provided a helper function `refineF` taking in F and the two sets of points, which you can call from `eight_point` before unscaling F.

- Remember that the x-coordinate of a point in the image is its column entry and y-coordinate is the row entry. Also note that eight-point is just a "figurative" name, it just means that you need at least 8 points; your algorithm should use an over-determined system ($N > 8$ points).

- To visualize the correctness of your estimated F, use the provided function `displayEpipolarF`, which takes in F, and the two images. This GUI lets you select a point in one of the images and visualize the corresponding epipolar line

in the other image (Figure 1).

- Please include in your report the recovered F and the visualization of some epipolar lines (similar to Figure 1).

- Helper functions:

In [ ]:
```
# helper function 1: singualrizes F using SVD
def _singularize(F):
    U, S, V = np.linalg.svd(F)
    S[-1] = 0
    F = U.dot(np.diag(S).dot(V))

    return F

# helper function 2.1: defines an objective function using F and the epipolar constraint
def _objective_F(f, pts1, pts2):
    F = _singularize(f.reshape([3, 3]))
    num_points = pts1.shape[0]
    hpts1 = np.concatenate([pts1, np.ones([num_points, 1])], axis=1)
    hpts2 = np.concatenate([pts2, np.ones([num_points, 1])], axis=1)
    Fp1 = F.dot(hpts1.T)
    FTp2 = F.T.dot(hpts2.T)

    r = 0
    for fp1, fp2, hp2 in zip(Fp1.T, FTp2.T, hpts2):
        r += (hp2.dot(fp1))**2 * (1/(fp1[0]**2 + fp1[1]**2) + 1/(fp2[0]**2 + fp2[1]**2))

    return r

# helper function 2.2: refines F using the objective from above and local optimization
def refineF(F, pts1, pts2):
    f = scipy.optimize.fmin_powell(
        lambda x: _objective_F(x, pts1, pts2), F.reshape([-1]),
        maxiter=100000,
        maxfun=10000
    )

    return _singularize(f.reshape([3, 3]))
```

- Visualization functions:

In [ ]:
```
# helper function 3.1: derives the epipoles using the essential matrix
def _epipoles(E):
    U, S, V = np.linalg.svd(E)
    e1 = V[-1, :]
    U, S, V = np.linalg.svd(E.T)
    e2 = V[-1, :]

    return e1, e2

# helper function 3.2: GUI that uses F to draw the epipolar lines in I2 correponding to chosen pts in I1
def displayEpipolarF(I1, I2, F):
    e1, e2 = _epipoles(F)

    sy, sx, _ = I2.shape

    f, [ax1, ax2] = plt.subplots(1, 2, figsize=(12, 9))
    ax1.imshow(I1)
    ax1.set_title('Select a point in this image')
    ax1.set_axis_off()
    ax2.imshow(I2)
    ax2.set_title('Verify that the corresponding point \n is on the epipolar line in this image')
    ax2.set_axis_off()

    while True:
        plt.sca(ax1)
        x, y = plt.ginput(1, mouse_stop=2)[0]
```

```
        xc, yc = int(x), int(y)
        v = np.array([[xc], [yc], [1]])

        l = F @ v
        s = np.sqrt(l[0]**2+l[1]**2)

        if s==0:
            error('Zero line vector in displayEpipolar')

        l = l / s
        if l[1] != 0:
            xs = 0
            xe = sx - 1
            ys = -(l[0] * xs + l[2]) / l[1]
            ye = -(l[0] * xe + l[2]) / l[1]
        else:
            ys = 0
            ye = sy - 1
            xs = -(l[1] * ys + l[2]) / l[0]
            xe = -(l[1] * ye + l[2]) / l[0]

        ax1.plot(x, y, '*', MarkerSize=6, linewidth=2)
        ax2.plot([xs, xe], [ys, ye], linewidth=2)
        plt.draw()
```

- Example result:

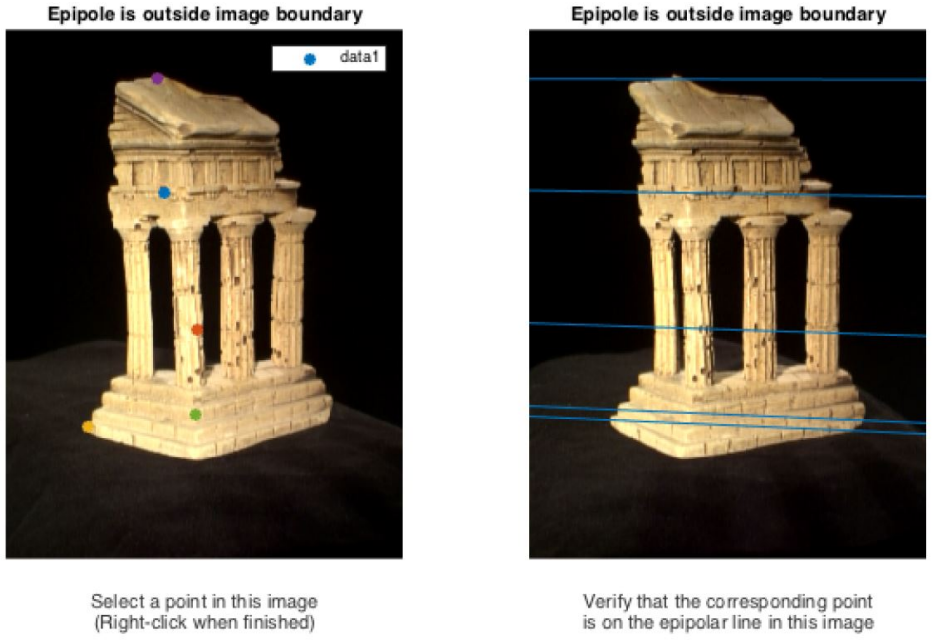


Figure 1 - Epipolar lines visualization from "displayEpipolarF"

1.2 - Epipolar Correpondences

To reconstruct a 3D scene with a pair of two images, we need to find many point pairs. A point pair is two points (one in each image) that correspond to the same 3D scene point. With enough of these pairs, when we plot the resulting 3D points, we will have a rough outline of the 3D object. You found point pairs in HW1 using feature detectors and feature descriptors, and testing a point in one image with every single point in the other image. But here we can use the fundamental matrix to greatly simplify this search.

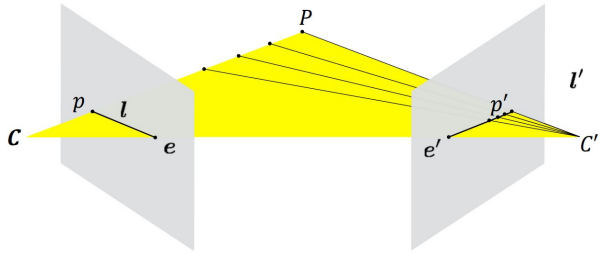


Figure 2 - Epipolar Geometry. Potential matches for p lie on the epipolar line l'.

Recall from class that given a point p in one image (the left view in Figure 2). Its corresponding 3D scene point P could lie anywhere along the line from the camera center C to the point p. This line, along with a second image's camera center C' (the right view in Figure 2) forms a plane. This plane intersects with the image plane of the second camera, resulting in a line l' in the second image which describes all the possible locations that p may be found in the second image. Line l' is the epipolar line, and we only need to search along this line to find a match for point p found in the first image.

- Write the following function:

In [ ]:
```
def epipolar_correspondences(I1, I2, F, pts1):
    """
    Epipolar Correspondences
    [I] I1, image 1 (H1xW1 matrix)
        I2, image 2 (H2xW2 matrix)
        F, fundamental matrix from image 1 to image 2 (3x3 matrix)
        pts1, points in image 1 (Nx2 matrix)
    [O] pts2, points in image 2 (Nx2 matrix)
    """
    # replace pass by your implementation
    pass
```

where `I1` and `I2` are two-view images, `F` is the fundamental matrix computed for the two images using your `eight_point` function, `pts1` is a $N \times 2$ matrix containing the (x, y) points in the first image, and the function should return `pts2`, a $N \times 2$ matrix, which contains the corresponding points in the second image. Implementation tips:

- To match one point p in image 1, use the fundamental matrix to estimate the corresponding epipolar line l' and generate a set of candidate points in the second image.
- For each candidate points p', a similarity score between p and p' is computed. The point among candidates with highest score is treated as epipolar correspondence.
- There are many ways to define the similarity between two points. Feel free to use whatever you want and **describe it in your write-up**. One possible solution is to select a small window of size w around the point p. Then compare this target window to the window of the candidate point in the second image. For the images we gave you, simple Euclidean distance or Manhattan distance should suffice.
- Remember to take care of data type and index range. You can use the provided function `epipolarMatchGUI` to visually test your function. Your function does not need to be perfect, but it should get most easy points correct, like corners, dots etc.
- Please include a screenshot of `epipolarMatchGUI` running with your implementation of `epipolar_correspondences` (similar to Figure 3). Mention the similarity metric you decided to use. Also comment on any cases where your matching algorithm consistently fails, and why you might think this is.

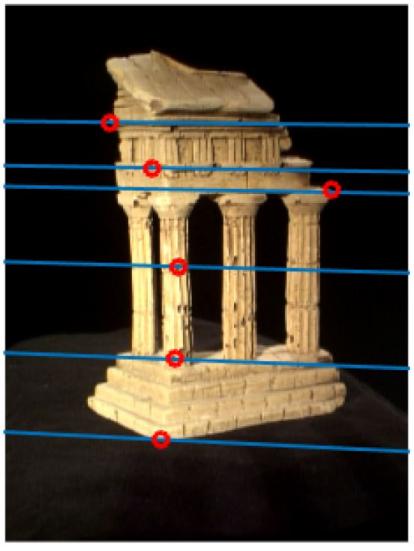
- Visualization function:

In [ ]:
```
def epipolarMatchGUI(I1, I2, F):
    e1, e2 = _epipoles(F)
```

```
    sy, sx, sd = I2.shape

    f, [ax1, ax2] = plt.subplots(1, 2, figsize=(12, 9))
    ax1.imshow(I1)
    ax1.set_title('Select a point in this image')
    ax1.set_axis_off()
    ax2.imshow(I2)
    ax2.set_title('Verify that the corresponding point \n is on the epipolar line in this image')
    ax2.set_axis_off()

    while True:
        plt.sca(ax1)
        x, y = plt.ginput(1, mouse_stop=2)[0]

        xc, yc = int(x), int(y)
        v = np.array([[xc], [yc], [1]])

        l = F @ v
        s = np.sqrt(l[0]**2+l[1]**2)

        if s==0:
            error('Zero line vector in displayEpipolar')

        l = l / s
        if l[0] != 0:
            xs = 0
            xe = sx - 1
            ys = -(l[0] * xs + l[2]) / l[1]
            ye = -(l[0] * xe + l[2]) / l[1]
        else:
            ys = 0
            ye = sy - 1
            xs = -(l[1] * ys + l[2]) / l[0]
            xe = -(l[1] * ye + l[2]) / l[0]

        ax1.plot(x, y, '*', MarkerSize=6, linewidth=2)
        ax2.plot([xs, xe], [ys, ye], linewidth=2)

        # draw points
        pc = np.array([[xc, yc]])
        p2 = epipolar_correspondences(I1, I2, F, pc)
        ax2.plot(p2[0,0], p2[0,1], 'ro', MarkerSize=8, linewidth=2)
        plt.draw()
```

- Example result:

Select a point in this image
(Right-click when finished)



Verify that the corresponding point
is on the epipolar line in this image

Figure 3 - Epipolar Match visualization. A few errors are alright, but it should get most easy points correct (corners, dots, etc.)

1.3 - Essential Matrix

In order to get the full camera projection matrices we need to compute the Essential matrix. So far, we have only been using the Fundamental matrix.

- Write the following function:

In [ ]:
```
def essential_matrix(F, K1, K2):
    """
    Essential Matrix
    [I] F, the fundamental matrix (3x3 matrix)
        K1, camera matrix 1 (3x3 matrix)
        K2, camera matrix 2 (3x3 matrix)
    [O] E, the essential matrix (3x3 matrix)
    """
    # replace pass by your implementation
    pass
```

Where `F` is the Fundamental matrix computed between two images, `K1` and `K2` are the intrinsic camera matrices for the first and second image respectively (contained in `data/intrinsics.npz`), and `E` is the computed essential matrix. The intrinsic camera parameters are typically acquired through camera calibration. Refer to the class slides for the relationship between the Fundamental matrix and the Essential matrix.

- Please include your estimated E matrix for the temple image pair in the PDF report.

1.5 - Triangulation

Write a function to triangulate pairs of 2D points in the images to a set of 3D points.

- Write the following function:

In [ ]:
```
def triangulate(M1, pts1, M2, pts2):
    """
    Triangulation
    [I] M1, camera projection matrix 1 (3x4 matrix)
        pts1, points in image 1 (Nx2 matrix)
        M2, camera projection matrix 2 (3x4 matrix)
        pts2, points in image 2 (Nx2 matrix)
    [O] pts3d, 3D points in space (Nx3 matrix)
    """
    # replace pass by your implementation
    pass
```

Where `pts1` and `pts2` are the $N \times 2$ matrices with the 2D image coordinates, `M1` and `M2` are the 3×4 camera projection matrices and `pts3d` is an $N \times 3$ matrix with the corresponding 3D points (in all cases, one point per row). Remember that you will need to multiply the given intrinsic matrices with your solution for the extrinsic camera matrices to obtain the final camera projection matrices.

- For `M1` you can assume no rotation or translation, so the extrinsic matrix is just $[I|0]$.
- For `M2`, pass the essential matrix to the provided function `camera2` to get four possible **extrinsic** matrices. You will need to determine which of these is the correct one to use (see hint in next section). Refer to the class slides to remind yourself of the 4 possible configurations.
- Keep in mind to multiply the extrinsics matrices by the corresponding intrinsics before inputting it to `triangulate`.

- Once implemented, check the performance by looking at the re-projection error. To compute the re-projection error, project the estimated 3D points back to the image 1 and compute the mean Euclidean error between projected 2D points and the given `pts1`.

- **In your write-up**: Describe how you determined which extrinsic matrix is correct. Note that simply rewording the hint is not enough. Report your re-projection error using the given `pts1` and `pts2` in `data/some_corresp.npz`. If implemented correctly, the re-projection error should be less than 2 pixels.

- Helper functions:

In [ ]:
```
def camera2(E):
    U,S,V = np.linalg.svd(E)
    m = S[:2].mean()
    E = U.dot(np.array([[m,0,0], [0,m,0], [0,0,0]])).dot(V)
    U,S,V = np.linalg.svd(E)
    W = np.array([[0,-1,0], [1,0,0], [0,0,1]])

    if np.linalg.det(U.dot(W).dot(V))<0:
        W = -W

    M2s = np.zeros([3,4,4])
    M2s[:,:,0] = np.concatenate([U.dot(W).dot(V), U[:,2].reshape([-1, 1])/abs(U[:,2]).max()], axis=1)
    M2s[:,:,1] = np.concatenate([U.dot(W).dot(V), -U[:,2].reshape([-1, 1])/abs(U[:,2]).max()], axis=1)
    M2s[:,:,2] = np.concatenate([U.dot(W.T).dot(V), U[:,2].reshape([-1, 1])/abs(U[:,2]).max()], axis=1)
    M2s[:,:,3] = np.concatenate([U.dot(W.T).dot(V), -U[:,2].reshape([-1, 1])/abs(U[:,2]).max()], axis=1)

    return M2s
```

1.6 - Putting It All Together

You now have all the pieces you need to generate a full 3D reconstruction. Write a test script `test_temple_coords` that does the following:

- Load the two images and the point correspondences from `data/some_corresp.npz`.
- Run eight point to compute the fundamental matrix `F`.
- Load the points in image 1 contained in `data/temple_coords.npz` and run your `epipolar_correspondences` on them to get the corresponding points in image 2.
- Load `data/intrinsics.npz` and compute the essential matrix `E`.
- Compute the first camera projection matrix `M1` and use `camera2` to compute the four candidates for `M2`.
- Run your `triangulate` function using the four sets of camera matrix candidates (after scaling with the intrinsics), the points from `data/temple_coords.npz`, and their computed correspondences.
- Figure out the correct `M2` and the corresponding 3D points. Hint: You'll get 4 projection matrix candidates for camera2 from the essential matrix. The correct configuration is the one for which most of the 3D points are in front of both cameras (positive depth), and make up a reasonable shape.
- Use matplotlib's scatter function to plot these point correspondences on screen.
- Report your computed extrinsic parameters (`R1` , `R2` , `t1` , `t2`) in your PDF.
- We will use your test script to run your code, so be sure it runs smoothly. In particular, use relative paths to load files, not absolute paths.
- **In your write-up**: Include 3 images of your final reconstruction of the points given in the file `data/temple_coords.npz`, from different angles as shown in Figure 4.

- Example result:

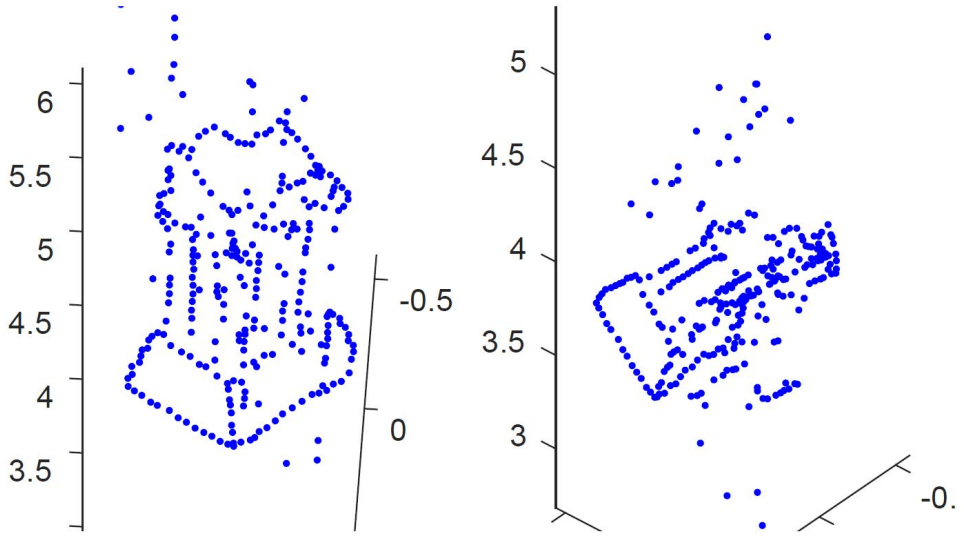


Figure 4 - Sample Reconstructions.

Part 2 - Pose Estimation (Bonus)

In this section, you will implement what you have learned in class to estimate the intrinsic and extrinsic parameters of camera given 2D points p on image and their corresponding 3D points P. In other words, given a set of matched points $\{P_i, p_i\}$ and camera model (in homogeneous coordinates):

$$p = MP$$

we want to find the estimate of the camera matrix $M \in \mathbb{R}^{3\times4}$, as well as intrinsic parameter matrix $K \in \mathbb{R}^{3\times3}$, camera rotation $R \in \mathbb{R}^{3\times3}$ and camera translation $t \in \mathbb{R}^{3\times1}$, such that:

$$M = K\,[R|t]$$

2.1 - Estimating M

- Write a function that estimates the camera matrix M given 2D and 3D points p, P.

In [ ]:
```
def estimate_pose(p, P):
    """
    Camera Matrix Estimation
    [I] p, 2D points (Nx2 matrix)
        P, 3D points (Nx3 matrix)
    [O] M, camera matrix (3x4 matrix)
    """
    # replace pass by your implementation
    pass
```

where `p` is $2 \times N$ matrix denoting the (x, y) coordinates of the N points on the image plane and `P` is $3 \times N$ matrix denoting the (x, y, z) coordinates of the corresponding points in the 3D world. Recall that this camera matrix can be computed using the same strategy as homography estimation by Direct Linear Transform (DLT).

- Once you finish this function, you can run the following script to test your implementation:

In [ ]:
```
# 1. Generate random camera matrix
K = np.array([[1,0,100], [0,1,100], [0,0,1]])
R, _,_ = la.svd(np.random.randn(3,3))
if la.det(R) < 0: R = -R
```

```
t = np.vstack((np.random.randn(2,1), 1))
M = K @ np.hstack((R, t))

# 2. Generate random 2D and 3D points
N = 100
P = np.random.randn(N,3)
p = M @ np.hstack((P, np.ones((N,1)))).T
p = p[:2,:].T / np.vstack((p[2,:], p[2,:])).T

# 3. Test pose estimation with clean points
Mc = estimate_pose(p, P)
pp = Mc @ np.hstack((P, np.ones((N,1)))).T
pp = pp[:2,:].T / np.vstack((pp[2,:], pp[2,:])).T

print('Reprojection Error with clean 2D points:', la.norm(p-pp))
print('Pose Error with clean 2D points:', la.norm((Mc/Mc[-1,-1])-(M/M[-1,-1])))

# 4. Test pose estimation with noisy points
p = p + np.random.rand(p.shape[0], p.shape[1])
Mn = estimate_pose(p, P)
pp = Mn @ np.hstack((P, np.ones((N,1)))).T
pp = pp[:2,:].T / np.vstack((pp[2,:], pp[2,:])).T

print('Reprojection Error with noisy 2D points:', la.norm(p-pp))
print('Pose Error with noisy 2D points:', la.norm((Mn/Mn[-1,-1])-(M/M[-1,-1])))
```

- **In your write-up**: Please include the output of this script in your PDF.

2.2 - Intrinsic/Extrinsic Parameters

- Write a function that estimates both intrinsic and extrinsic parameters from a camera matrix:

In [ ]:
```
def estimate_params(M):
    """
    Camera Parameter Estimation
    [I] M, camera matrix (3x4 matrix)
    [O] K, camera intrinsics (3x3 matrix)
        R, camera extrinsics rotation (3x3 matrix)
        t, camera extrinsics translation (3x1 matrix)
    """
    # replace pass by your implementation
    pass
```

From what we learned on class, the `estimate_params` function should consecutively run the following steps:

- Compute the camera center c by using SVD. Hint: c is the eigenvector corresponding to the smallest eigenvalue.
- Compute the intrinsic K and rotation R by using QR decomposition. K is a right upper triangle matrix while R is an orthonormal matrix.
- Compute the translation by $t = -Rc$.

- Once you finish this function, you can run the following script to test your implementation:

In [ ]:
```
# 1. Generate random camera matrix
K = np.array([[1,0,100], [0,1,100], [0,0,1]])
R, _,_ = la.svd(np.random.randn(3,3))
if la.det(R) < 0: R = -R
t = np.vstack((np.random.randn(2,1), 1))
M = K @ np.hstack((R, t))

# 2. Generate random 2D and 3D points
N = 100
P = np.random.randn(N,3)
p = M @ np.hstack((P, np.ones((N,1)))).T
p = p[:2,:].T / np.vstack((p[2,:], p[2,:])).T
```

```
# 3. Test parameter estimation with clean points
Mc = estimate_pose(p, P)
Kc, Rc, tc = estimate_params(Mc)

print('Intrinsic Error with clean 2D points:', la.norm((Kc/Kc[-1,-1])-(K/K[-1,-1])))
print('Rotation Error with clean 2D points:', la.norm(R-Rc))
print('Translation Error with clean 2D points:', la.norm(t-tc))

# 4. Test parameter estimation with noisy points
p = p + np.random.rand(p.shape[0], p.shape[1])
Mn = estimate_pose(p, P)
Kn, Rn, tn = estimate_params(Mn)

print('Intrinsic Error with noisy 2D points:', la.norm((Kn/Kn[-1,-1])-(K/K[-1,-1])))
print('Rotation Error with noisy 2D points:', la.norm(R-Rn))
print('Translation Error with noisy 2D points:', la.norm(t-tn))
```

- **In your write-up**: Please include the output of this script in your PDF.

2.3 - Projecting CAD Models to 2D

Now you will utilize what you have implemented to estimate the camera matrix from a real image, shown in Figure 5 (left), and project the 3D object (CAD model), shown in Figure 5 (right), back on to the image plane.

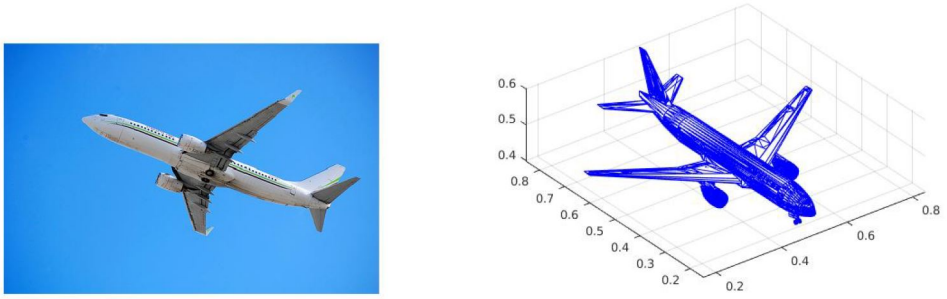


Figure 5 - The provided image and 3D CAD model.

Write a script that does the following:

- Load an image `image`, a CAD model `cad`, 2D points `x` and 3D points `X` from the data file `data/pnp.npz`.
- Run `estimate_pose` and `estimate_params` to estimate camera matrix M, intrinsic matrix K, rotation matrix R, and translation t.
- Use your estimated camera matrix M to project the given 3D points P onto the image.
- Plot the given 2D points p and the projected 3D points on screen. An example is shown in Figure 6 (left).
- Draw the CAD model rotated by your estimated rotation R on screen. An example is shown in Figure 6 (middle).
- Project the CAD's all vertices onto the image and draw the projected CAD model overlapping with the 2D image. An example is shown in Figure 6 (right).
- **In your write-up**: Please include the three images similar to Figure 6 in your PDF. You must use different colors from Figure 6. For example, green circle for given 2D points, black points for projected 3D points, blue CAD model, and red projected CAD model overlapping on the image. You will get **NO credit** if you use the same color.

- Example result:

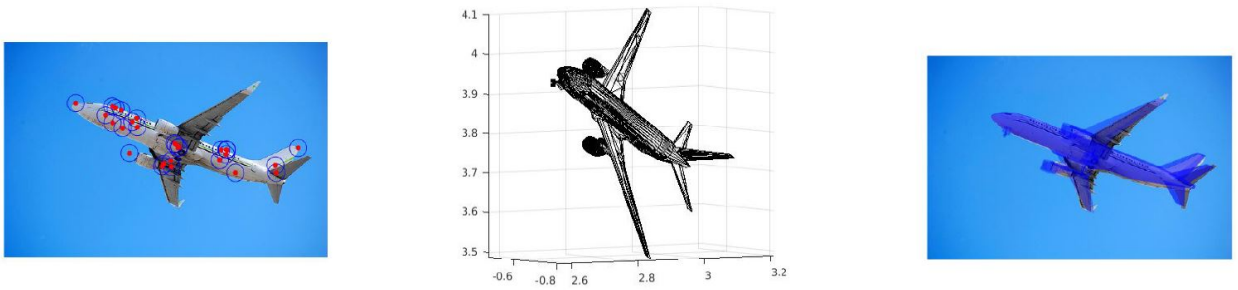


Figure 6 - Project a CAD model back onto the image. Left: the image annotated with given 2D points (blue circle) and projected 3D points (red points); Middle: the CAD model rotated by estimated R; Right: the image overlapping with projected CAD model.

References & Credits

- Carnegie Mellon University - CMU
- Icons from Icon8.com - https://icons8.com