Principles of Robot Autonomy I
Homework 3
Due Friday, October 30 (11:59pm)

You will submit your homework to Gradescope. Your submission will consist of a single pdf with your answers for written questions and relevant plots from code.

Your submission must be typeset in LaTeX.

Problem 1: Chessboard Camera Extrinsics

Starter code for this question has been made available online through GitHub. To get started, download the code by running `git clone https://github.com/StanfordASL/AA174a-HW3.git` in a terminal window.

NOTE: In this homework, we'll be using a very popular open-source computer vision package named OpenCV. This is reflected in the `requirements.txt` file in the homework repository.

Introduction

Objective: In this problem you will solve one of the fundamental problems in computer vision, computing the position of a camera relative to a known marker arrangement. In this case, the markers will be the corners of the squares on a checkerboard with known dimensions. In this problem you will learn how cameras can be used for localization against features, and how projection to and from the image plane into world coordinates works!

Setup: This entire problem is contained in the `p1_extrinsics.ipynb` Jupyter notebook. We recommend developing in the VM to mitigate any dependency issues, but feel free to develop locally if you are comfortable setting up a python environment and installing the necessary dependencies.

Background:

Camera Extrinsics: The position $\mathbf{t} \in \mathbb{R}^3$ and orientation $\mathbf{R} \in SO(3)$ of a camera with respect to some "world" frame are often jointly referred to as the camera *extrinsic* parameters, or *extrinsics* for short. Once obtained, the extrinsics can be used to transform a 3D point in the world frame $\mathbf{P} \in \mathbb{R}^3$ into a 3D point in the camera frame $\mathbf{P'} \in \mathbb{R}^3$ as follows:

$$\mathbf{P'} = \mathbf{R}\mathbf{P} + \mathbf{t} \tag{1}$$

Notice that in Eq. 1, $\mathbf{R}$ and $\mathbf{t}$ actually represent the position and orientation of the world frame with respect to the camera frame. The goal of this problem is to compute $\mathbf{R}$ and $\mathbf{t}$ of a camera for a world frame that lies on the surface of a chessboard.

Chessboard: Everything in this problem starts with a single image of a checkerboard. Without adding any code you can run through the first two few cells in the Jupyter notebook to see what image we are working with for this problem. We use a checkerboard because we can easily physically measure the side-lengths of the squares (for our problem the squares have a dimension of 0.0205m). Furthermore, the black and white

pattern of the checkerboard has high contrast and it makes it easy to find the features that we are going to use for calibration, square "cross-junctions".

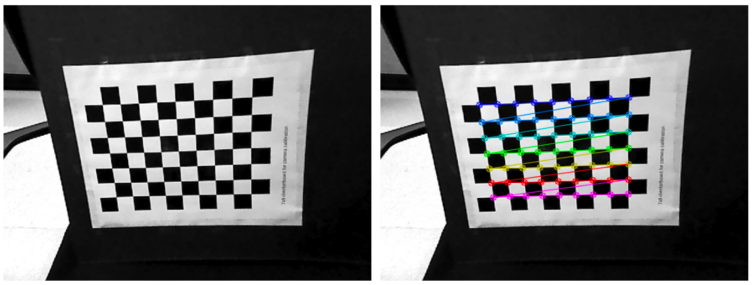


Figure 1: Cross-junction (corner) extraction of chessboards.

Feature Extraction

The first step towards computing the pose of the camera relative to the checkerboard is accurately estimating the position of the cross-junctions in the image frame. That is, in the image of the checkerboard, which pixels correspond to the cross-junctions? Fortunately, for us modern roboticists this question was answered quite a while ago by researchers studying computer vision. The solution is to apply a series of transformations on the image that isolate the high-contrast regions of the squares on the board, and then to use the Harris Corner Detection Algorithm to detect the cross-junctions. This combination of methods is provided in the `cv2.findChessboardCorners(...)` function provided by OpenCV.

(i) ⌨ Use the `cv2.findChessboardCorners(...)` along with the corner parameters specified in the code to extract the cross-junction pixel locations, and plot them on top of the image using the `ax.plot(...)` function. Include this plot in your write-up.

Extrinsic Calibration

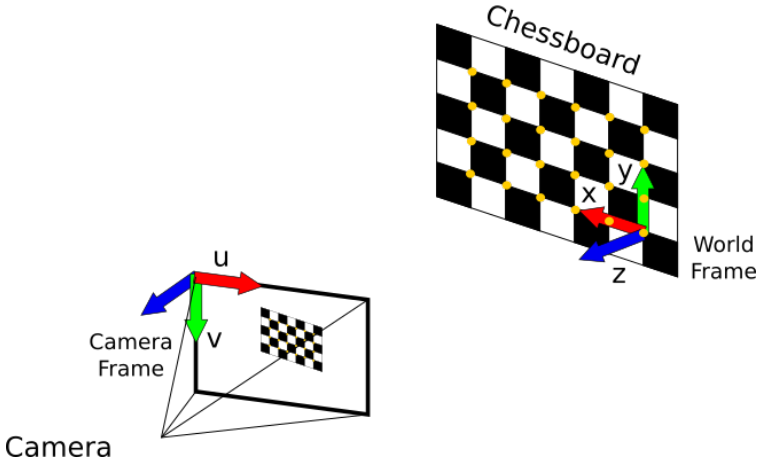


Figure 2: Chessboard problem setup.

In this part of the problem you will use a known camera intrinsics matrix $\mathbf{K}$, the known distance between cross-junctions (0.0205 m), and the cross-junction image coordinates to compute the relative position and rotation of the camera to the chessboard.

To solve for the position and rotation of the camera relative to the chessboard you will make use of a method that is similar to the solution that was presented for the PnP problem discussed in Lecture 8. The main difference here is that all of the points exist on the surface of the chessboard, causing all of their z-components

to be zero in the world frame, see Figure 2. Inspecting the equation for transforming world coordinates into homogeneous pixel coordinates we can see that this results in a slightly different set of equations:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \propto \mathbf{K}[\mathbf{R}\ \mathbf{t}] \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \mathbf{K}[\mathbf{r_0}\ \mathbf{r_1}\ \mathbf{r_2}\ \mathbf{t}] \begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix} = \mathbf{K}[\mathbf{r_0}\ \mathbf{r_1}\ \mathbf{t}] \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = \mathbf{H} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} \tag{2}$$

where $\mathbf{r_0}$ is the first column of $\mathbf{R}$ and so on. We know the location of pixels in both image coordinates (due to cross-junction detection), and in world coordinates (from known chessboard geometry). Therefore, equation 2 forms a system of equations for each 2D-3D correspondence. For cross-junction i the resulting set of equations is

$$\begin{bmatrix} -X_i & -Y_i & -1 & 0 & 0 & 0 & u_iX_i & uY_i & u_i \\ 0 & 0 & 0 & -X_i & -Y_i & -1 & v_iX_i & v_iY_i & v_i \end{bmatrix} \begin{bmatrix} \mathbf{h_0}^\top \\ \mathbf{h_1}^\top \\ \mathbf{h_2}^\top \end{bmatrix} = \mathbf{M}_i \begin{bmatrix} \mathbf{h_0}^\top \\ \mathbf{h_1}^\top \\ \mathbf{h_2}^\top \end{bmatrix} = 0, \tag{3}$$

where $\mathbf{h_0} \in \mathbb{R}^{3\times1}$ is the first row of $\mathbf{H} \in \mathbb{R}^{3\times3}$. Forming $\mathbf{M_i} \in \mathbb{R}^{2\times9}$ for each of the points and stacking them vertically forms a new matrix $\mathbf{M}$ whose null-space corresponds to the flattened elements of $\mathbf{H}$. The null-space of a matrix can be found by computing the Singular Value Decomposition, and choosing the column of the $\mathbf{V}$ matrix with the smallest singular value.

After computing $\mathbf{H}$, using the known intrinsics matrix for the camera $\mathbf{K}$, the pose of the camera relative to the world frame can be computed as

$$\mathbf{K}^{-1}\mathbf{H} = \lambda[\mathbf{r_0}\ \mathbf{r_1}\ \mathbf{t}]. \tag{4}$$

The scale factor, λ, arises from the use of homogeneous coordinates. One way to compute it is to choose $\lambda = ||(\mathbf{K}^{-1}\mathbf{H})_{(:,0)}||_2$, or equal to the norm of the first column of the left hand side matrix in Equation 4. This corresponds to a scaling such that the first column of the resulting rotation matrix has a norm equal to 1. Finally, the third column of the rotation matrix must be orthoganol to the other two columns, and therefore $\mathbf{r_2} = \mathbf{r_0} \times \mathbf{r_1}$.

To summarize,

$$\mathbf{r_0} = \frac{1}{\lambda}(\mathbf{K}^{-1}\mathbf{H})_{(:,0)} \quad \mathbf{r_1} = \frac{1}{\lambda}(\mathbf{K}^{-1}\mathbf{H})_{(:,1)} \quad \mathbf{r_2} = \mathbf{r_0} \times \mathbf{r_1} \quad \mathbf{t} = \frac{1}{\lambda}(\mathbf{K}^{-1}\mathbf{H})_{(:,2)}. \tag{5}$$

(ii) Use the method outlined above, and the corner locations from (i) to compute the rotation, $\mathbf{R}$, and translation, $\mathbf{t}$, of the camera with respect to the chessboard in the image, and include the values that you compute in your write up with elements up to two significant figures.

World to Camera Projection

Using the camera intrinsic and extrinsic parameters, arbitrary points can be projected onto the camera frame from the world frame.

(iii) Complete the function `transform_world_to_camera`, and use it to project the world-frame chessboard corner locations onto the image along with the actual corner locations. Include the resulting plot in your write-up. Hint: you might find Equation 2 useful for this part.

Finally, run the last cell in the notebook to project a secret point cloud in world frame onto your chessboard. This is an example of how the methods in this problem can be used for augmented reality applications (no need to submit anything for this).

Problem 2: Linear Filtering

The field of computer vision includes processing complex and high dimensional data (up to millions of pixels per image) and extracting relevant features that can be used by other components in a robotic autonomy stack. Nowadays, many computer vision techniques rely on deep learning and machine learning algorithms for classification and depend heavily on computational tools that can efficiently process, learn, and do inference on the data. However, these methods can be quite computationally expensive to use (often requiring GPU acceleration). What else can we do if we have a limited computational budget? How did robots perceive and detect objects in their environments before the advent of machine learning?

Note: The indexing we use in this problem set differs slightly from what you've seen in lecture. The reason for this is one of practicality, it is much easier to handle generally-sized filters in the manner we use in this problem set. Mathematically, the exact same operation is performed, the only thing that changes is the coordinate at which we perform the correlation denotes the top-left element of the filter and image region. Additionally, as a result the output image will be shifted up and to the left, but this is remedied by indexing into the output in the exact same way (referring to the top-left coordinate).

(i) Let us analyze how grayscale filters affect grayscale images when correlated with them.

Given an image $I \in \mathbb{R}^{m\times n}$ represented as a $m \times n$ matrix, where each element $I(i,j) \in [0, 255]$ denotes the grayscale value of a pixel and the indices $i = 0, \dots, m-1$ and $j = 0, \dots, n-1$ denote the position of the corresponding pixel, the **correlation** operator $G = F \otimes I$ produces an image whose pixels $G(i,j)$ are a sum of the original image I's pixels weighted by the entries in a given filter $F \in \mathbb{R}^{k\times\ell}$[1]. Specifically, given a filter $F \in \mathbb{R}^{k\times\ell}$, the output image $G \in \mathbb{R}^{m\times n}$ is defined pixelwise as

$$G(i,j) = \sum_{u=0}^{k-1}\sum_{v=0}^{\ell-1} F(u,v)\cdot \bar{I}(i+u, j+v), \tag{6}$$

where $\bar{I} \in \mathbb{R}^{(m+k-1)\times(n+\ell-1)}$ is the original image I padded with zeros along its edges. In this problem, we consider the image $I \in \mathbb{R}^{3\times3}$ (and its corresponding zero-padded image $\bar{I} \in \mathbb{R}^{5\times5}$)

$$I = \begin{bmatrix} 7 & 4 & 1 \\ 8 & 5 & 2 \\ 9 & 6 & 3 \end{bmatrix}, \quad \text{with } \bar{I} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 7 & 4 & 1 & 0 \\ 0 & 8 & 5 & 2 & 0 \\ 0 & 9 & 6 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

In the following, manually compute the output image $G \in \mathbb{R}^{3\times3}$ as the result of $G = F \otimes I$ when the filter F is:

(a) $F = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

(b) $F = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

(c) $F = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$.

- For part (c), also add a one-sentence explanation as to what this filter is doing to the image, and why this might be useful in computer vision.

[1] We assume k and ℓ are odd numbers for notational convenience.

- Let, $F' = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$. How do you think filter F is different from filter F' in its functionality? (You don't need to calculate G for F'.)

If you cannot answer this right away, come back to this after you've implemented correlation and run your code with this filter!

(d) $F = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$.

- For part (d), also add a one-sentence explanation as to what this filter is doing to the image, and why this might be useful in computer vision.
- Let, $F' = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$. How do you think filter F is different from filter F' in its functionality? (You don't need to calculate G for F'.)

If you cannot answer this right away, come back to this after you've implemented correlation and run your code with this filter!

For all the above parts, please include the matrix G in your write-up.

(ii) This entire sub-problem is contained in the `p2_correlation.ipynb` Jupyter notebook. Please see the setup instructions at the top of this document.

- **Part 1:** Implement the `correlate_image` function in `p2_correlation.ipynb`. This function takes as input a grayscale image and a filter, performs the correlation operation (as shown in Eq. 6), and returns the filtered image.
  - Run the evaluation code beneath `correlate_image` to perform the correlation operation on the image using a horizontal edge detection filter and a vertical edge detection filter. Include the resulting images in your submission.
- **Part 2:** Implement the `create_gaussian_filter` function in `p2_correlation.ipynb`. This function takes as input the standard deviation σ for the Gaussian filter, and constructs a Gaussian filter in $\mathbb{R}^{3\times3}$.
  - Run the evaluation code beneath `create_gaussian_filter` to perform the correlation operation on the image using Gaussian filters with $\sigma = 0.5$ and $\sigma = 2.0$. Include the resulting images in your submission and explain the effect of σ.