

Homework 5

EECS/BioE/MechE C106A/206A
Introduction to Robotics

Due: October 11, 2022

Note: This problem set includes programming components. Your deliverables for this assignment are:

1. A PDF file submitted to the HW5 (pdf) Gradescope assignment with all your work and solutions to the written problems.
2. The provided python files submitted to the HW5 (code) Gradescope assignment with your implementation of the programming components.

Theory

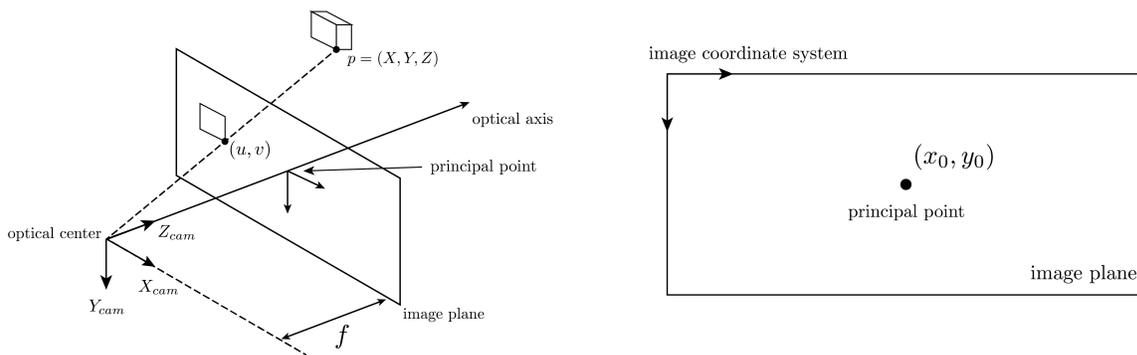


Figure 1: Geometry behind a Pinhole Camera

Homogeneous coordinates

In class, we have encountered homogeneous coordinates for 3D points, which work by appending a 1 to the end of the 3 vector to get a vector in 4D. When writing the coordinates of a point in the image plane (which is 2 dimensional) we will use *2D homogeneous coordinates*. This means we will represent the point $x = (u, v) \in \mathbb{R}^2$ as $(u, v, 1)$ in homogeneous coordinates. Note that the homogeneous representation of a 2D point is a 3D vector.

Image formation

We consider a pinhole model of image formation. See Figure 1. We denote the center of the perspective projection (the point in which all the rays intersect) as the optical center or camera center and the line perpendicular to the image plane passing through the optical center as the optical axis. Additionally, the intersection point of the image plane with the optical axis is called the principal point.

We always associate a reference frame with each camera as shown. By convention, we center this reference frame at the optical center, take the $X - Y$ plane of this reference frame to be parallel to the image plane, and take the Z -axis to be perpendicular to the image plane, pointing in the direction of viewing. Additionally, there is a 2D reference frame attached to the image plane, with respect to which the "image coordinates" of any point are measured. A discretized version of this reference frame give us the familiar "pixel coordinates" of any points (in columns and rows).

The figure shows a point p with spatial coordinates $\bar{X} = (X, Y, Z)$ in the camera reference frame, and image coordinates $x = (u, v)$. As we stated above, we will default to representing image coordinates in homogeneous form as $x = (u, v, 1)$, and usually we will overload this notation wherever it is obvious if we are using homogeneous or regular coordinates.

The camera parameters (such as the focal length f and others; see Lab 6) are specified in the form of a 3×3 *camera matrix* K . This matrix K is always invertible. The spatial coordinates \bar{X} (in the camera reference frame) and the image (homogeneous) coordinates $x = (u, v, 1)$ of a given point p are related via the K matrix as

$$x = \frac{1}{Z} K \bar{X} \quad (1)$$

where Z is the Z -coordinate of the point in the camera reference frame. Observe that this Z -coordinate has a significant geometric meaning. It is the distance from the camera's $X - Y$ plane to the point, along the direction of viewing. In other words, it is the "depth" of the point as seen from the camera. So, we give this depth its own symbol λ and move it to the LHS to get the less unwieldy expression

$$\lambda x = K \bar{X} \quad (2)$$

Note that given the depth λ , the camera matrix K , and the image coordinates x , the spatial coordinates \bar{X} of the point can be recovered by inverting equation (2). On the other hand, without knowing the depth, the spatial coordinates \bar{X} can only be recovered up to a scale factor. This makes geometric sense, since we can see from figure (1) that any point along the line connecting the optical center to p gets projected to the same image coordinates as p , and hence knowing only the image coordinates, we can at best specify a line along which p must lie.

Problem 1. Two-View Triangulation

Consider two cameras with reference frames $\{1\}$ and $\{2\}$ respectively. As always, the reference frame of each camera is such that the $X - Y$ plane is parallel to the image plane and the Z -axis points in the direction of viewing.

Assume we know the relative transform $g_{21} = (R, T) \in SE(3)$. Additionally, assume the cameras are calibrated and normalized, so that the camera matrix K is the identity.

Both cameras are looking at the same point p in 3D space, which has unknown coordinates $X_1 \in \mathbb{R}^3$ in frame $\{1\}$ and $X_2 \in \mathbb{R}^3$ in frame $\{2\}$. We observe their image coordinates x_1 and x_2 , written in 2D homogeneous coordinates.

- (a) Write down an expression relating x_1 to X_1 in terms of an unknown depth λ_1 . Do the same for camera 2.
- (b) Write down an expression for X_2 in terms of X_1 .
- (c) Find a method for solving for X_1 in terms of the known quantities $R \in SO(3), T \in \mathbb{R}^3, x_1, x_2$. Can you deal with the case when the image measurements x_1, x_2 are corrupted by some small (white, zero mean, Gaussian) noise?

Hint: Eliminate X_1 and X_2 from your expressions, and try to find only the unknown depths λ_1 and λ_2 . Then, use these depths to recover X_1 .

Problem 2. Epipolar Ambiguities and Structure from Motion

Consider a similar set up as in the previous problem, with two calibrated, normalized cameras, where the transform $g_{21} = (R, T)$ between them is *not* known. Recall that for such a system, we define $E = \hat{T}R \in \mathbb{R}^{3 \times 3}$ to be the *essential matrix*. The essential matrix imposes the *epipolar constraint*, which is that whenever x_1 and x_2 are the (homogeneous) image coordinates of the *same* point, then they must satisfy

$$x_2^T E x_1 = 0$$

Such a pair of image points x_1, x_2 that correspond to the same point in 3D space viewed from two different cameras are called *corresponding points*. In this problem, we consider the problem of recovering the relative poses between cameras in a multi-camera setup when we are given a number of corresponding-point pairs.

It turns out that 8 pairs of corresponding points $(x_1^{(1)}, x_2^{(1)}), \dots, (x_1^{(8)}, x_2^{(8)})$ in general position are enough to compute a candidate essential matrix \tilde{E} . Each such pair gives us an equation of the form

$$x_2^{(i)T} \tilde{E} x_1^{(i)} = 0 \tag{3}$$

where the x 's are all known. We additionally have the constraint that E should be of the correct form to be written as $\hat{T}R$. i.e. we should be able to write it as the product of a cross product matrix $\in \mathfrak{so}(3)$ and a rotation matrix $\in SO(3)$. We can then solve this system of equations for a nonzero 3×3 matrix E that satisfies this set of constraints. See chapter 5 from *An Invitation to 3D Vision (Ma, Soatto, Kosecka, Sastry)* for the full details.

- (a) Show that we can only recover E up to a scale factor. In particular, show that if \tilde{E} is a matrix that satisfies all the required constraints, then so is $c\tilde{E}$ for any real number c .

Remark: We can in fact conclude that this ambiguity can be attributed to an unknown scale factor on the translation vector T between the two frames. This means that although we can decompose a computed essential matrix E into rotational and translational components (R, \tilde{T}) , we can only recover the original translation T up to a scale factor. Typically then, we restrict ourselves to finding a \tilde{T} such that $\|\tilde{T}\| = 1$.

- (b) Say we have a system of 3 cameras with reference frames $\{1\}, \{2\}$ and $\{3\}$ respectively, and we are able to recover the transforms $(R_{12}, \tilde{T}_{12}), (R_{23}, \tilde{T}_{23})$ and (R_{13}, \tilde{T}_{13}) using point correspondances, where each \tilde{T}_{ij} has norm 1. So there are unknown, nonzero scale factors λ_{ij} such that the true $T_{ij} = \lambda_{ij}\tilde{T}_{ij}$. If we could find the three scalars $\lambda_{12}, \lambda_{23}, \lambda_{13}$ then we would have fully recovered the relative poses between the various cameras. Show that in this setting, we can only recover the λ_{ij} 's up to a single scaling factor.
- (c) Consider the same setup as part (b), but now the translation T_{12} is known exactly (i.e. λ_{12} is known). Show that now, all λ_{ij} 's can be recovered and the relative poses between the cameras can be found.

Problem 3. Planar Motion Models

For feature tracking algorithms we often assume that the motion of points in the image when restricted to a small window can be approximated through different *transformations* of varying levels of complexity. These assumptions may only hold for a small window, but for an appropriate object, there exist motions in 3D space such that these transformations are accurate over the entire image. In this question, we will determine what those motions are.

Assume we are only concerned with the motion of image points corresponding to an object where all points in the object have the same z -coordinate z_o relative to the camera frame (ie. all world points of interest lie in some plane parallel to the $x - y$ plane which passes through the point $[0 \ 0 \ z_o]^T$)

- (a) Define $h(x)$ as a function which maps an image point to its new location after the corresponding world point undergoes a rigid motion. Let's consider a scenario where we measure image motion $h(x)$ and we notice that each point on the image corresponding to our object translates by the same Δx . More concretely, $h(x) = x + \Delta x$. Prove that a rigid body motion $R = I$ and $T = [a \ b \ 0]^T$ applied to our object corresponds to this $h(x)$.
- (b) Now let's consider a scenario where $h(x) = Ax + d$ for image points corresponding to our object. Prove that a rigid body motion $R = R_z(\theta)$ with arbitrary T applied to our object corresponds to this motion.

Problem 4. I'm On Point Cloud Nine

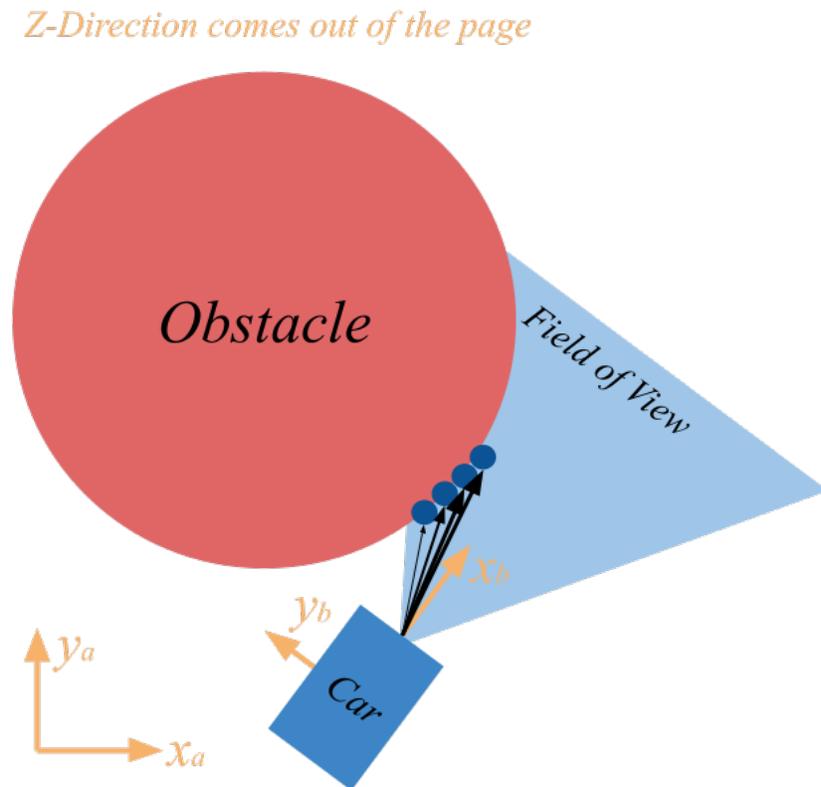
In this problem, we'll tackle a practical problem in robotics: obstacle detection and avoidance! In this question, we'll look at the obstacle detection and avoidance problem through the lens of *autonomous driving*. Let's begin!

In this problem, we've provided you with a simulation environment that models the motion of a simple vehicle. Let's briefly discuss the different files, their purposes within the larger simulation environment, and the basics of how they work. Note that in this question, we'll only ask you to interact with three of these files, so don't worry about understanding all of the code - much of it has already been implemented for you!

1. **dynamics.py**: This file specifies everything about the motion of the vehicle. It uses a physics-based model to tell us how the vehicle's state will change depending on the inputs we give it.
2. **state_estimation.py**: Although the dynamics file tells us everything about how the vehicle moves, in the real world, we find out information about systems through noisy *sensors*. We can use sensors such as motion capture systems, accelerometers, and encoders to gain a strong *estimate* of where our car is located in space. In the real world, and in our simulation, sensors always have some *noise* - they will never give us an exact reading for the state of our vehicle!
In addition to caring about the state of our car, we know that the state of the environment is important as well! Our car also has a depth camera that allows it to read the position of obstacles in the environment.
3. **trajectory.py**: Suppose we want to give our vehicle a goal position that we'd like it to reach. Is it enough to give it a single position, or is it better to supply it with a trajectory - a set of positions as a function of time? This class gives some simple straight-line trajectories between points in N dimensions. All of these trajectories are differentiable, and have zero start and end velocity for smoothness.
4. **obstacle.py**: This file keeps track of the obstacles in the environment - what their shape is and where they're located. It also holds `ObstacleQueue`, an interface between the physical obstacles and the noisy sensor data about the position of obstacles that's read by the depth camera.
5. **controller.py**: Given the state of the vehicle, a trajectory we'd like the vehicle to follow, and an estimate of where the obstacles are in space, how can we figure out what inputs to send to the vehicle at each point in time? This file defines a set of *feedback controllers* that allow the vehicle to plan around obstacles and track trajectories. Our vehicle uses a special type of feedback controller called a control barrier function quadratic program (CBF-QP) to safely avoid the obstacles.
6. **lyapunov_barrier.py** This file contains functions known as control barrier functions and control Lyapunov functions, which are vital to the operation of the obstacle-avoiding feedback controller.

7. **environment.py** This file manages the simulation of the vehicle in time. It uses two concepts - one from the field of numerical analysis and one from control theory - called Euler integration and Zero Order Hold to simulate the behavior of the vehicle. It also includes some utilities to analyze the behavior of the vehicle.
8. **run_simulation.py** This file is the master file that sets up and runs the vehicle simulation. This is the only file you'll actually need to run in this question!

Let's take a closer look at the scenario of this problem. We have an autonomous car that we'd like to drive to some position in space, but there's an obstacle in the way!



Above: the car avoids the obstacle and successfully navigates to its goal

To understand where the obstacle is located, the car has a sensor known as a **depth camera**. This is a sensor that's rigidly attached to the car. It senses the position of points on the surfaces of obstacles. Since the depth camera is rigidly attached to the car, all of the obstacle point positions it detects are specified in the *car frame*. We call the entire collection of obstacle points detected by the depth camera a **pointcloud**.

In this question, we'll process the pointcloud collected by the depth camera, and turn it from a collection of points in the car frame to a collection of points that are useful for autonomous trajectory planning.

Running your code

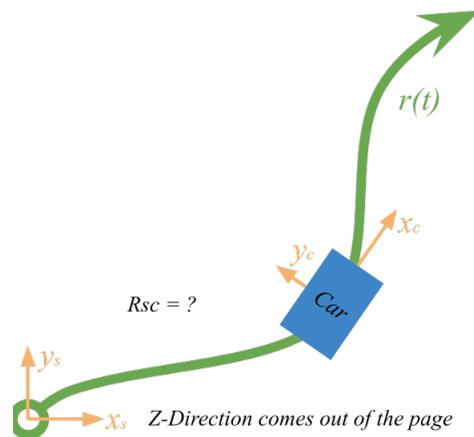
Before you run your code, make sure you have all of the necessary python libraries installed. Run the following commands in your command line to ensure you have them all installed:

1. pip install numpy
2. pip install matplotlib
3. pip install scipy
4. pip install casadi

To run your code, all you need to do is run the file `run_simulation.py`. Give this a try to see what the vehicle does before processing the pointcloud! By the end of this problem, your vehicle will successfully avoid the obstacle.

Questions:

- (a) **Finding car orientation:** To figure out where the obstacle actually is in the spatial frame, we'll need to figure out the orientation of our car. However, our car only has a limited range of sensors! All we know about the car is its 3D position in the world frame and its 3D velocity in the world frame. How can we use just these two pieces of information to figure out its orientation?



Here's what we know: the first basis vector of the car frame, x_c , always points tangent to the direction of motion. The z -basis vector of the car frame, z_c is always parallel to the z -basis vector of the spatial frame, z_s .

Using this information, derive a formula for the rotation matrix from the car frame to the spatial frame, R_{sc} .

Once you have your formula, open the file `state_estimation.py` and go to the function `get_orient()`, on line 62. When completed, this function will return the rotation matrix of the car.

In this function, you are provided with the car position (`carPos`) and velocity (`carVel`) in the spatial frame as 3×1 NumPy arrays. Using this information, complete the function so it returns the rotation matrix R_{sc} . Note that when you first open the file,

there will be a placeholder value for the rotation matrix - your final answer should change this placeholder!

Hint: Consider the velocity vector of the vehicle - how can this help us find the first basis vector of our car frame?

- (b) **Transforming the pointcloud:** Now that we have the orientation of the car, we wish to transform the pointcloud of obstacle points into the world frame, where we can use it for path planning and obstacle avoidance.

Go to line 77 of the file `obstacle.py`, to the function called `depth_to_spatial()`. This function takes in a pointcloud called `ptMatrix`, in the car frame, and returns that pointcloud transformed into spatial frame.

`ptMatrix` is a $3 \times N$ NumPy array. Each column of the matrix represents an individual point sensed by the depth camera in the car frame.

With this information, complete the function so that it returns `transformedPoints`, a $3 \times N$ NumPy array where each column is the corresponding column of `ptMatrix`, transformed into the spatial frame.

This will give us our whole pointcloud in the spatial frame!

Hint: look at the function `np.tile()`. You can use it to complete this question in a single line!

- (c) **K Nearest Neighbors:** We almost have everything we need! Our final step is to restrict the pointcloud to the most important points. Real depth cameras will read up to 400000 points at once! This is far too many for us to work with at any one time. Instead of working with such an enormous point cloud, we'll focus on the set of the K points in the pointcloud that are closest to the vehicle.

Go to the file `state_estimation.py`, and scroll down to line 172, where there is a function called `get_knn()`. This function takes in an integer K , which is the number of points we wish to find in our pointcloud. In the first line of this function, we extract the pointcloud sensed by the vehicle, as seen from the vehicle frame. This variable, "`ptcloud`," is a $3 \times N$ NumPy array, where each column is a point in the pointcloud.

We now want to find the K points in this pointcloud that are closest to the vehicle. Note that since the pointcloud is already in the vehicle frame, this problem is equivalent to solving for the K points in the pointcloud closest to the zero vector, which is the position of the vehicle in the vehicle frame. Complete the function so that it returns `closest_K`, a $3 \times K$ matrix of the points in the pointcloud closest to the vehicle. Note that these points should be in the vehicle frame, *not* the spatial frame.

Your solution to this problem, called the *K Nearest Neighbors* problem, should just use NumPy, and no external libraries. Note that at the moment, this function calls an efficient implementation of the KNN algorithm using the library SciPy. You should replace this implementation with your own!

These are the only three functions you'll need to implement! Once your car is avoiding the obstacle after completing all three functions, you're all good to go! Remember, the only file you need to run is `run_simulation.py`.

When your implementation is correct, you'll see a trajectory that looks like the following:

