Hardware for Self-driving Cars

A. Giavaras

Contents

1 Hardware for Self-driving Cars

In this chapter, we will discuss sensors, and the various types of them available for the task of perception. Next we will discuss the self-driving car hardware available nowadays.

1.1 Sensors

Let's begin by talking about sensors. Even the best perception algorithms are limited by the quality of their sensor data. And careful selection of sensors can go a long way to simplifying the self-driving perception task. Let's try to give a definition of what a sensor is.

Definition 1.1. What is a sensor?

For our purposes, a sensor is any device that measures or detects some property of the environment, or changes to that property over time.

Sensors are broadly categorized into two types, depending on what property they record. If they record a property of the environment they are called **exteroceptive**. Extero means outside, or from the surroundings. On the other hand, if the sensors record a property of the ego vehicle, they are called **proprioceptive**. Proprios means internal, or one's own. Let's start by discussing common exteroceptive sensors.

1.1.1 Camera

We start with the most common and widely used sensor in autonomous driving, the camera. Cameras are a passive, light-collecting sensor that are great at capturing rich, detailed information about a scene. In fact, some groups believe that the camera is the only sensor truly required for self-driving. But state of the art performance is not yet possible with vision alone. While talking about cameras, we usually tend to talk about three important comparison metrics. We select cameras in terms:

- resolution
- field of view or FOV
- dynamic range

The resolution is the number of pixels that create the image. So it's a way of specifying the quality of the image. The field of view is defined by the horizontal and vertical angular extent that is visible to the camera, and can be varied through lens selection and zoom. The dynamic range of the camera is the difference between the darkest and the lightest tones in an image. High

dynamic range is critical for self-driving vehicles due to the highly variable lighting conditions encountered while driving especially at night.

There is an important trade off cameras and lens selection, that lies between the choice of field of view and resolution. Wider FOV permits a lager viewing region in the environment, but fewer pixels that absorb light from one particular object. As the FOV increases, we need to increase resolution to still be able to perceive with the same quality, the various kinds of information we may encounter. Other properties of cameras that affect perception exist as well, such as focal length, depth of field and frame rate.

The combination of two cameras with overlapping fields of view and aligned image planes is called the stereo camera. Stereo cameras allow depth estimation from synchronized image pairs. Pixel values from image can be matched to the other image producing a disparity map of the scene. This disparity can then be used to estimate depth at each pixel.

1.1.2 LIDAR

Next we have LIDAR which stands for light detection and ranging sensor. LIDAR sensing involves shooting light beams into the environment and measuring the reflected return. By measuring the amount of returned light and time of flight of the beam. Both in intensity in range to the reflecting object can be estimated. An illustration of LIDAR based environment representation is shown in figure 1.

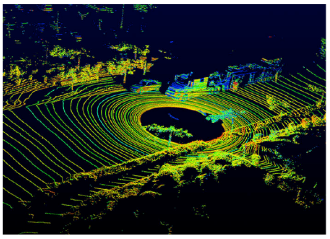


Fig. 1: LIDAR illustration of environment representation.

LIDAR usually includes a spinning element with multiple stacked light sources and outputs a three dimensional point cloud map, which is great for assessing scene geometry. Because it is an active sensor with it's own light sources, LIDAR are not effected by the environments lighting. So LIDAR do not face the same challenges as cameras when operating in poor or variable lighting conditions. Let's discuss the important comparison metrics for selecting LIDAR.

- The first is the number of sources it contains with 8, 16, 32, and 64 being common sizes.

- the second is the points per second it can collect. The faster the point collection, the more detailed the 3D point cloud can be.

Another characteristic is the rotation rate. The higher this rate, the faster the 3D point clouds are updated. Detection range is also important, and is dictated by the power output of the light source. And finally, we have the field of view, which once again, is the angular extent visible to the LIDAR sensor.

Finally, we should also mention the new LIDAR types that are currently emerging. High-resolution, solid-state LIDAR. Without a rotational component of the typical LIDARs, these sensors stand to become extremely low-cost and reliable. Thanks to being implemented entirely in silicon. HD solid-state LIDAR are still a work in progress. But definitely something exciting for the future of affordable self-driving.

1.1.3 Radar

Our next sensor is RADAR, which stands for radio detection and ranging. RADAR sensors have been around longer than LIDAR and robustly detect large objects in the environment. They are particularly useful in adverse weather as they are mostly unaffected by precipitation. Let's discuss some of the comparison metrics for selecting RADAR. RADAR are selected based on

- detection range
- field of view,
- the position and speed measurement accuracy.

RADARs are also typically available as either having a wide angular field of view but short range. Or having a narrow FOV but a longer range.

1.1.4 Ultrasonics or sonars

The next sensor we are going to discuss are ultrasonics or sonars. Originally so named for sound navigation and ranging. Which measure range using sound waves. Sonars are sensors that are short range and inexpensive ranging devices. This makes them good for parking scenarios, where the ego-vehicle needs to make movements very close to other cars. Another great thing about sonar is that they are low-cost. Moreover, just like RADAR and LIDAR, they are unaffected by lighting and precipitation conditions. A sonar sensor is selected based on a few key metrics itemized next.

- The maximum range they can measure
- The the detection FOV
- The cost

1.1.5 Proprioceptive sensors

Now let's discuss the proprioceptive sensors, the sensors that sense ego properties. The most common ones here are:

- Global Navigation Satellite Systems, GNSS for short, such as GPS or Galileo
- Inertial Measurement Units or IMU's
- Wheel odometers

GNSS receivers are used to measure ego vehicle position, velocity, and sometimes heading. The accuracy depends a lot on the actual positioning methods and the corrections used. Apart from these, the IMU also measures the angular rotation rate, accelerations of the ego vehicle, and the combined measurements can be used to estimate the 3D orientation of the vehicle. Where heading is the most important for vehicle control. Finally, we have wheel odometry sensors. This sensor tracks the wheel rates of rotation, and uses these to estimate the speed and heading rate of change of the ego car. This is the same sensor that tracks the mileage on your vehicle.

In summary, the major sensors used nowadays for autonomous driving perception include cameras, RADAR, LIDAR, sonar, GNSS, IMUs, and wheel odometry modules. These sensors have many characteristics that can vary wildly, including resolution, detection range, and FOV.

Selecting an appropriate sensor configuration for a self-driving car is not trivial. Figure 2 is a simple graphic that shows each of the sensors and where they usually go on a car.

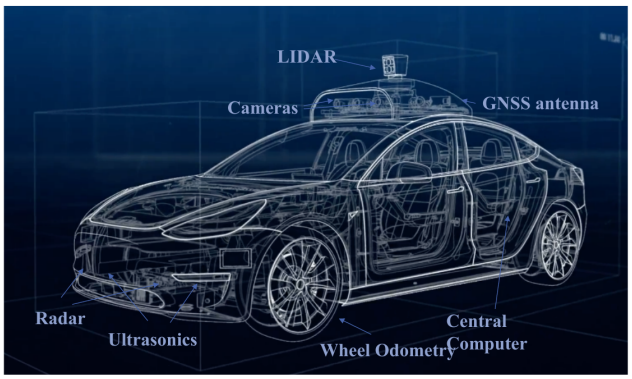


Fig. 2: Usual sensor positioning.

1.2 Computing Hardware

Let's now discuss a little bit about the computing hardware most commonly used in self-driving cars at the time of writing. The most crucial part is the computing brain, the main decision making unit of the car. It takes in all sensor data and outputs the commands needed to drive the vehicle. Most companies prefer to design their own computing systems that match the specific requirements of their sensors and algorithms. Some hardware options exist, however, that can handle self-driving computing loads out of the box.

The most common examples would be Nvidia's Drive PX and Intel & Mobileye's EyeQ. Any computing brain for self-driving needs both serial and parallel compute modules. Particularly for image and LIDAR processing to do segmentation, object detection, and mapping. For these we employ GPUs, FPGAs and custom ASICs (Application Specific Integrated Circuit), which are specialized hardware to do a specific type of computation.

For example, the drive PX units include multiple GPUs. The EyeQs have FPGAs both to accelerate parallalizable compute tasks, such as image processing or neural network inference.

Finally, a quick comment about synchronization. Because we want to make driving decisions based on a coherent picture of the road scene. It is essential to correctly synchronize the different modules in the system, and serve a common clock. Fortunately, GPS relies on extremely accurate timing to function, and as such can act as an appropriate reference clock when available. Regardless, sensor measurements must be timestamped with consistent times for sensor fusion to function correctly. Let's summarize. In this video, we learned about sensors and their different types based on what they measure.

1.3 Hardware Configuration

Section 1.1, covers the various kinds of sensors most commonly used for perception. One question that should be answered is how do we place these sensors on the vehicle in order to aquire a complete view of the environment?

In this section, we will discuss the configuration design to meet sensor coverage needs for an autonomous driving car. We will do this by going through two common scenarios:

- Driving on a highway and
- Driving in an urban environment

After analyzing these scenarios, we will lay out the overall coverage requirements and discuss some issues with the design.

Let's however begin by recalling the most commonly available sensors. These are:

- The camera for appearance input.
- The stereo camera for depth information
- Lidar for all whether 3D input
- Radar for object detection
- Ultrasonic for short-range 3D input
- GNSS/IMU data and wheel odometry for ego state estimation.

Also, remember that all of these sensors come in different configurations and different ranges in FOV over which they can sense. They have some resolution that depends on the instrument specifics and the field of view. Before we move to discussing coverage, let's define the deceleration rates we're willing to accept for driving which will drive the detection ranges needed for our sensors.

Remark 1.1. **Aggressive Deceleration**

Aggressive deceleration is set to $5m/sec^2$ which is roughly the deceleration you experience when you slam the brakes hard and try to stop abruptly in case of an emergency.

Normal decelerations are set to $2m/sec^2$, which is reasonably comfortable while still allowing the car to come to a stop quickly. Given a constant deceleration our braking distance d can be computed as follows according to equation 1.

$$d = \frac{V^2}{2\alpha} \tag{1}$$

where V is the vehicle velocity and α is its rate of deceleration. We can also factor in reaction time of the system and road surface friction limits, but we'll keep things simple in this discussion.

Let's talk about coverage now. The question we want to answer is where should we place our sensors so that we have sufficient input for our driving task? Practically speaking, we want our sensors to capture the ODD we have in mind or the ODD our system can produce decisions for. We should be able to provide all of the decisions with sufficient input. There can be so many possible scenarios in driving but we'll look at just two common scenarios to see how the requirements drive our sensor selection. Will look at highway and urban driving. Let's think about these two situations briefly.

For a divided highway, we have fast moving traffic, usually high volume, and quite a few lanes to monitor, but all vehicles are moving in the same direction. The other highlight of driving on a highway setting is that there are fewer and gradual curves and we have exits and merges to consider as well.

On the other hand, in the urban situation we'll consider, we have moderate volume and moderate speed traffic with fewer lanes but with traffic moving in all directions especially through intersections.

1.3.1 Highway Scenario

Let's start with the highway setting. We can break down the highway setting into three basic maneuver needs.

- We may need to hit the brakes hard if there's an emergency situation.
- We need to maintain a steady speed matching the flow of traffic around us.
- We might need to change lanes.

In the case of an emergency stop, if there is a blockage on our road we want to stop in time. So, applying our stopping distance equation longitudinally, we need to be able to sense about a 110 meters in front of us assuming a highway speed of a 120 kilometers and aggressive deceleration. Most self-driving systems aim for sensing ranges of a 150 to 200 meters in front of the vehicle as a result. Similarly, to avoid lateral collision or to change lanes to avoid hitting an obstacle in our lane, we need to be able to sense at least our adjacent lanes, which are 3.7 meters wide in North America. To maintain speed during vehicle following, we need to sense the vehicle in our own lane. Both their relative position and the speed are important to maintain a safe following distance. This is usually defined in units of time for human drivers and set to two seconds in nominal conditions. It can also be assessed using aggressive deceleration of the lead vehicle and the reaction time from our ego vehicle. So, at a 120 kilometers per hour, relative position and speed measurements to a range of 165 meters are needed and typical systems use 100 meters for this requirement. Laterally, we need to know what's happening anywhere in our adjacent lanes in case another vehicles seeks to merge into our lane or we need to merge with other traffic. A wide 160 to 180 degree field of view is required to track adjacent lanes and a range of 40 to 60 meters is needed to find space between vehicles.

Finally, let's discuss the lane change maneuver and consider the following scenario. Suppose we want to move to the adjacent lane, longitudinally we need to look forward, so we are a safe distance from the leading vehicle and we also need to look behind just to see what the rear vehicles are doing and laterally it's a bit more complicated. We may need to look beyond just the adjacent lanes. For example, what if a vehicle attempts to maneuver into the adjacent lane at the same time as we do? We'll need to coordinate our lane change room maneuvers so we don't crash.

The sensor requirements for lane changes are roughly equivalent to those in the maintain speed scenario. As both need to manage vehicles in front of and behind the ego vehicle as well as to each side. Overall, this gives us the picture for

coverage requirements for the highway driving scenario. We need longitudinal sensors and lateral sensors and both wide and narrow FOV sensors to do these three maneuvers, the emergency stop, maintaining speed and changing lanes. Already from this small set of ODD requirements we see a large variety of sensor requirements that arise.

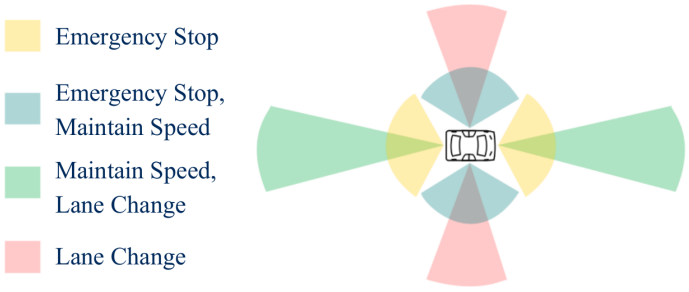


Fig. 3: Highway analysis overall coverage.

1.3.2 Urban Scenario

Let's discuss the urban scenario next. The urban scenario as we discussed before is a moderate volume, moderate traffic scenario with fewer lanes on the highway case but with the added complexity of pedestrians. There are six types of basic maneuvers here. Obviously, we can still perform emergency stop, maintain speed and lane changes but we also have scenarios such as overtaking a parked car, left and right turns at intersections and more complex maneuvers through intersections such as roundabouts. In fact, for the first three basic maneuvers, the coverage analysis is pretty much the same as the highway analysis but since we are not moving as quickly, we don't need the same extent for our long-range sensing.

Let's discuss the overtake maneuver next. More specifically, consider a case where you have to overtake a parked car. Longitudinally, we definitely need to sense the parked car as well as look for oncoming traffic. So, we need both sensors, wide short-range sensors to detect the parked car and narrow long-range sensors to identify if oncoming traffic is approaching. Laterally, we'll need to observe beyond the adjacent lanes for merging vehicles as we did in the highway case. Intersections require that we have near omni-directional sensing for all kinds of movements that can occur. Approaching vehicles, nearby pedestrians, doing turns and much more. Finally, for roundabouts we need a wide-range, short distance sensor laterally since the traffic is slow but we also need a wide-range short distance sensor longitudinally because of how movement around the

roundabout occurs. We need to sense all of the incoming traffic flowing through the roundabout to make proper decisions.

Thus, we end up with the overall coverage diagram for the urban case shown in figure 4. The main difference with respect to highway coverage is because of the sensing we require for movement at intersections and at roundabouts and for the overtaking maneuver. In fact, the highway case is almost entirely covered by the urban requirements.

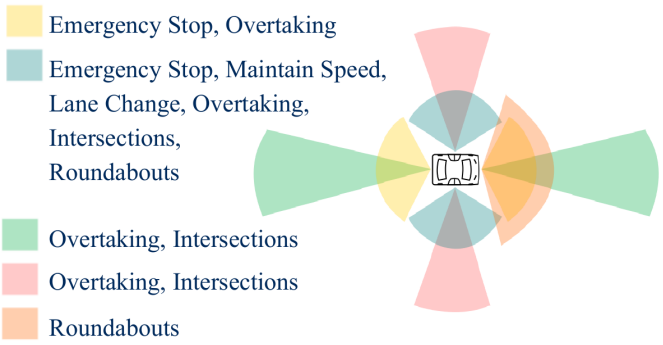


Fig. 4: Urban analysis overall coverage.

Let's summarize the coverage analysis. For all of the maneuvers we do, we need long range sensors which typically have shorter angular field of view and wide angular field of view sensors which typically have medium to short-range sensing. As the scenarios become more complex, we saw the need for full 360 degrees sensor coverage on the short scale out to about 50 meters and much longer range requirements in the longitudinal direction. We can also add even shorter range sensors like sonar which are useful in parking scenarios and so in the end our sensor configuration looks something like this diagram.

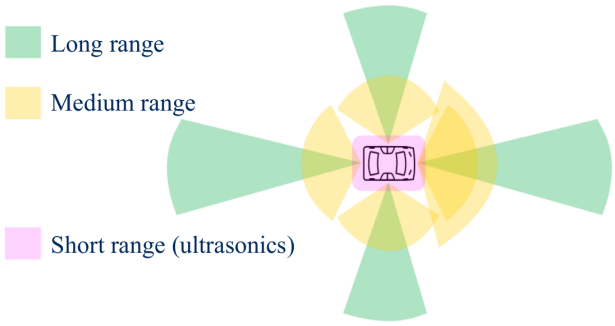


Fig. 5: Overall coverage.

To summarize, our choice of sensors should be driven by the requirements of the maneuvers we want to execute and it should include both long-range sensors for longitudinal dangers and wide field of view sensors for omnidirectional perception. The final choice of configurations also depends on our requirements for operating conditions, sensor redundancy due to failures and on budget. There is no single answer to which sensors are needed for a self-driving car.

2 Questions

1. What are the differences between exteroceptive sensors and proprioceptive sensors? (Select all that apply)
   (a) Proprioceptive sensors do not interact with the environment, whereas exteroceptive sensors do.
   (b) Proprioceptive sensors can determine distance traveled by the vehicle, whereas exteroceptive sensors cannot.
   (c) Exteroceptive sensors can determine obstacle size and distance away, whereas proprioceptive sensors cannot.
   (d) Proprioceptive sensors are used to determine vehicle position, whereas exteroceptive sensors are used for sensing the environment.
   (e) Exteroceptive sensors can determine distance traveled by the vehicle, whereas proprioceptive sensors cannot.
2. Which of the following exteroceptive sensors would you use in harsh sunlight?
   (a) Sonar
   (b) Cameras
   (c) Radar
   (d) Lidar
3. Why is synchronization and timing accuracy important in the self driving system? Choose the primary reason.
   (a) Synchronization is important to ensure that sensors measure the environment at the same time.
   (b) Synchronization is important to ensure organized computation.
   (c) Synchronization is important to ensure correct sensor fusion.
   (d) Synchronization is important to check sensor failure.
4. Your autonomous vehicle is driving on the German autobahn at $150km/h$ and you wish to maintain safe following distances with other vehicles.

Assuming a safe following distance of $2s$, what is the distance (in m) required between vehicles? Round your answer to 2 decimal places.

5. Using the same speed of $150km/h$, what is the braking distance (in m) required for emergency stops? Assume an aggressive deceleration of $5m/s^2$. Round your answer to two decimal places.

6. Suppose your vehicle was using long range cameras for sensing forward distance, but it is now nighttime and the images captured are too dark. Which of the following sensors can be used to compensate?

   (a) Lidar

   (b) IMU

   (c) Radar

   (d) Sonar

7. What are the differences between an occupancy grid and a localization map? (Select all that apply)

   (a) The localization map is primarily used to estimate the vehicle position, whereas the occupancy grid is primarily used to plan collision free paths.

   (b) The localization map uses only lidar data, whereas the occupancy grid can use both lidar and camera data.

   (c) An occupancy grid uses a dense representation of the environment, whereas a localization map does not need to be dense.

   (d) The occupancy grid only contains static objects, while the localization map contains only dynamic objects.

8. The vehicle steps through the software architecture and arrives at the controller stage. What information is required for the controller to output its commands to the vehicle?

   (a) Planned paths

   (b) Vehicle state

   (c) Locations of obstacles and other vehicles

   (d) Environment maps

9. What is (are) the role(s) of the system supervisor? (Select all that apply)

   (a) To ensure that the maps update at the correct frequencies

   (b) To ensure that the planned paths are collision free

   (c) To ensure that the controller outputs are within operating range

   (d) To ensure that the sensors are working correctly

10. Which of the following tasks should be assigned to the local planner?
    (a) Planning a merge onto the highway
    (b) Planning a route to a destination
    (c) Planning a lane change to turn left
    (d) Planning to avoid a parked car in the ego vehicle's lane
11. What common objects in the environment appear in the occupancy grid?
    (a) Other moving vehicles
    (b) Lane boundaries
    (c) Parked vehicles
    (d) Traffic lights
12. Which of the following maps contain roadway speed limits?
    (a) Occupancy grid
    (b) Localization map
    (c) Detailed roadmap

3 Sensor Calibration

Now that we've seen how we can combine multiple sources of sensor data to estimate the vehicle state, it's time to address the topic of **sensor calibration**. Sensor calibration and is absolutely essential for doing state estimation properly. Concretely, in this section, we will discuss the three main types of sensor calibration and why we need to think about them when designing a state estimator for a self-driving car. The three main types of calibration will talk about are

- Intrinsic calibration, which deals with sensors specific parameters
- Extrinsic calibration, which deals with how the sensors are positioned and oriented on the vehicle
- Temporal calibration, which deals with the time offset between different sensor measurements.

3.1 Intrinsic calibration

Let's look at intrinsic calibration first. In intrinsic calibration, we want to determine the fixed parameters of our sensor models, so that we can use them in

an estimator like an extended Kalman filter. Every sensor has parameters associated with it that are unique to that specific sensor and are typically expected to be constant.

For example, we might have an encoder attached to one axle of the car that measures the wheel rotation rate ω. If we want to use ω to estimate the forward velocity v of the wheel, we would need to know the radius R of the wheel, so that we can use the following equation

$$v = \omega R \tag{2}$$

In this case, R is a parameter of the sensor model that is specific to the wheel the encoder is attached to and we might have a different R for a different wheel. Another example of an intrinsic sensor parameter is the elevation angle of a scan line in a LiDAR sensor like the Velodyne. The elevation angle is a fixed quantity but we need to know it ahead of time so that we can properly interpret each scan.

So, how do we determine intrinsic parameters like these? Well, there are a few practical strategies for doing this. The easiest one is just let the manufacturer do it for you. Often, sensors are calibrated in the factory and come with a spec sheet that tells you all the numbers you need to plug into your model to make sense of the measurements. This is usually a good starting point but it will not always be good enough to do really accurate state estimation because no two sensors are exactly alike and there will be some variation in the true values of the parameters. Another easy strategy that involves a little more work is to try measuring these parameters by hand. This is pretty straightforward for something like a tire, but not so straightforward for something like a LiDAR where it's not exactly practical to poke around with a protractor inside the sensor.

A more sophisticated approach involves estimating the intrinsic parameters as part of the vehicle state, either on the fly or more commonly as a special calibration step before putting the sensors into operation. This approach has the advantage of producing an accurate calibration that's specific to the particular sensor and can also be formulated in a way that can handle the parameters varying slowly over time.

For example, if you continually estimate the radius of your tires, this could be a good way of detecting when you have a flat. Now, because the estimators we've talked about in this course are general purpose, we already have the tools to do this kind of automatic calibration. In order to see how this works, let's come back to our example of a car moving in one dimension.

we have attached an encoder to the back wheel to measure the wheel rotation rate. If we want to estimate the wheel radius along with position and velocity, all we need to do is add it to the state vector and work out what the new motion and observation model should be. For the motion model, everything is the same

as before except now there's an extra row and column in the matrix that says that the wheel radius should stay constant from one time step to the next. For the observation model, we are still observing position directly through GPS but now we're also observing the wheel rotation rate through the encoder. So, we include the extra non-linear observation in the model. From here, we can use the extended or unscented Kalman filter to estimate the wheel radius along with the position and velocity of the vehicle.

Thus, intrinsic calibration is essential for doing state estimation with even a single sensor.

3.2 Extrinsic calibration

Extrinsic calibration is equally important for fusing information from multiple sensors. In extrinsic calibration, we are interested in determining the relative poses of all of the sensors usually with respect to the vehicle frame.

For example, we need to know the relative pose of the IMU and the LiDAR. The rates reported by the IMU are expressed in the same coordinate system as the LiDAR point clouds. Just like with intrinsic calibration, there are different techniques for doing extrinsic calibration. If you are lucky, you might have access to an accurate CAD model of the vehicle, where all of the sensor frames have been nicely laid out for you. If you are less lucky, you might be tempted to try measuring by hand. Unfortunately, this is often difficult or impossible to do accurately since many sensors have the origin of their coordinate system inside the sensor itself, and you probably do not want to dismantle your car and all of the sensors.

Fortunately, we can use a similar trick to estimate the extrinsic parameters by including them in our state. This can become a bit complicated for arbitrary sensor configurations, and there is still a lot of research being done into different techniques for doing this reliably.

3.3 Temporal calibration

Finally, an often overlooked but still important type of calibration is temporal calibration. In all of our discussion of multisensory fusion, we've been implicitly assuming that all of the measurements we have combined are captured exactly the same moment in time or at least close enough for a given level of accuracy. But how do we decide whether two measurements are close enough to be considered synchronized? Well, the obvious thing to do would just be to timestamp each measurement when the on-board computer receives it, and match up the measurements that are closest to each other.

For example, if we get LiDAR scans at 15 hertz and IMU readings at 200 hertz, we might want to pair each LiDAR scan with the IMU reading whose timestamp

is the closest match.

In reality, there is an unknown delay between when the LiDAR or IMU actually records an observation and when it arrives at the computer. These delays can be caused by the time it takes for the sensor data to be transmitted to the host computer, or by pre-processing steps performed by the sensor circuitry, and the delay can be different for different sensors. Hence, if we want to get a really accurate state estimate, we need to think about how well our sensors are actually synchronized, and there are different ways to approach this. The simplest and most common thing to do is just to assume the delay is zero. You can still get a working estimator this way, but the results may be less accurate than what you would get with a better temporal calibration.

Another common strategy is to use hardware timing signals to synchronize the sensors, but this is often an option only for more expensive sensor setups. As you may have guessed, it's also possible to try estimating these time delays as part of the vehicle state, but this can get complicated.

To summarize, sensor fusion is impossible without calibration. In this section, we touched upon three types of calibration. Intrinsic calibration, which deals with calibrating the parameters of our sensor models. Extrinsic calibration, which gives us the coordinate transformations we need to transform sensor measurements into a common reference frame. Temporal calibration, which deals with synchronizing measurements to ensure they all correspond to the same vehicle state. While there are some standard techniques for solving all of these problems, calibration is still very much an active area of research.

4 Questions

1. What is intrinsic calibration?
2. Why do we need extrinsic calibration?
3. What is temporal calibration?

5 LiDAR Principles

In this section, we will be talking about LIDAR, or light detection and ranging sensors. LIDAR has been an enabling technology for self-driving cars because it can see in all directions and is able to provide very accurate range information. In fact, with few exceptions, most self-driving cars on the road today are equipped with some type of LIDAR sensor. In this section, we will learn about the operating principles of LIDAR sensors, basic sensor models used to work with LIDAR data and LIDAR point clouds, different kinds of transformation operations applied to point clouds, and how we can use LIDAR to localize a self-driving car using a technique called point cloud registration.

Concretely, in this section, we will explore how a LIDAR works and take a look at sensor models for 2D and 3D LIDARs. We will also describe the sources of measurement noise and errors for these sensors.

5.1 Introduction

If you've ever seen a self-driving car like the Waymo vehicle or an Uber car, you've probably noticed something spinning on the roof of the car, see Figure 6. That something is a LIDAR, or light detection and ranging sensor, and its job is to provide detailed 3D scans of the environment around the vehicle.

Fig. 6: Various types of LiDAR sensors.

In fact, LIDAR is one of the most common sensors used on self-driving cars and many other kinds of mobile robots. LIDARs come in many different shapes and sizes, and can measure the distances to a single point, a 2D slice of the world, or perform a full 3D scan. Some of the most popular models used today are manufactured by firms such as Velodyne in California, Hokuyo in Japan, and SICK in Germany, see Figure 6. In this section, we will mainly focus on the Velodyne sensors as our example of choice, but the basic techniques applied to other types of LIDARs as well.

Remark 5.1. **Some History**

LIDAR was first introduced in the 1960s, not long after the invention of the laser itself. The first group to use LIDAR were meteorologists at the US National Center for Atmospheric Research, who deployed LIDAR to measure the height of cloud ceilings. These ground-based celiometers are still in use today not only to measure water clouds, but also to detect volcanic ash and air pollution. Airborne LIDAR sensors are commonly used today to survey and map the earth's surface for agriculture, geology,

military, and other uses. But the application that first brought LIDAR into the public consciousness was Apollo 15, the fourth manned mission to land on the moon, and the first to use a laser altimeter to map the surface of the moon.

5.2 LiDAR operating principles

So, we've seen that LIDAR can be used to measure distances and create a certain type of map, but how did they actually work, and how can we use them onboard a self-driving car? To build a basic LIDAR in one dimension, you need three components:

- A laser
- A photodetector
- A very precise stopwatch

The laser first emits a short pulse of light usually in the near infrared frequency band along some known ray direction. At the same time, the stopwatch begins counting. The laser pulse travels outwards from the sensor at the speed of light and hits a distant target. Maybe another vehicle in front of us on the road or a stationary object like a stop sign or a building. As long as the surface of the target is not too polished or shiny, the laser pulse will scatter off the surface in all directions, and some of that reflected light will travel back along the original ray direction. The photodetector catches that return pulse and the stopwatch tells you how much time has passed between when the pulse first went out and when it came back. That time is called the **round-trip** time.

Further, we know the speed of light, which is a bit less than 300 million meters per second. So, we can multiply the speed of light by the round-trip time to determine the total round trip distance traveled by the laser pulse Figure 7.

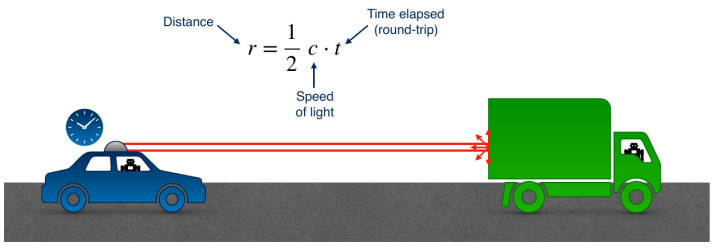


Fig. 7: Calculate distance with LiDAR.

Since light travels much faster than cars, it's a good approximation to think of the LIDAR and the target as being effectively stationary during the few nanoseconds that it takes for all of this to happen. That means that the distance from the LiDAR to the target is simply half of the round-trip distance we just calculated. This technique is called **time-of-flight ranging**. Although it's not the only way to build a LiDAR, it is a very common method that also gets used with other types of ranging sensors like radar and sonar. It is worth mentioning that the photodetector also tells you the intensity of the return pulse relative to the intensity of the pulse that was emitted. This intensity information is less commonly used for self-driving, but it provides some extra information about the geometry of the environment and the material the beam is reflecting off of.

It is possible to create 2D images from LiDAR intensity data that you can then use the same computer vision algorithms you'll learn about in the next course. Since LiDAR is its own light source, it actually provides a way for self-driving cars to see in the dark.

We know how to measure a single distance to a single point using a laser, a photodetector, a stopwatch, and the time-of-flight equation, but obviously it's not enough to stay laser focused on a single point ahead. How do we use this technique to measure a whole bunch of distances in 2D or in 3D? The trick is to build a rotating mirror into the LiDAR that directs the emitted pulses along different directions. As the mirror rotates, you can measure distances at points in a 2D slice around the sensor. If you then add an up and down nodding motion to the mirror along with the rotation, you can use the same principle to create a scan in 3D. For Velodyne type LiDARs, where the mirror rotates along the entire sensor body, it's much harder to use a knotting motion to make a 3D scan.

Instead, these sensors will actually create multiple 2D scan lines from a series of individual lasers spaced at fixed angular intervals, which effectively lets you paint the world with horizontal stripes of laser light. Figure 8 shows an example of a typical raw LiDAR stream from a Velodyne sensor attached to the roof of a car.

The black hole in the middle is a blind spot where the sensor itself is located, and the concentric circles spreading outward from there are the individual scan lines produced by the rotating Velodyne sensor. Each point in the scan is colored by the intensity of the return signal. The entire collection of points in the 3D scan is called a point cloud.

Now typically, LiDARs measure the position of points in 3D using spherical coordinates, range or radial distance from the center origin to the 3D point, elevation angle measured up from the sensors XY plane, and azimuth angle, measured counterclockwise from the sensors x-axis. This makes sense because the azimuth and elevation angles tell you the direction of the laser pulse, and the range tells you how far in that direction the target point is located. The azimuth and elevation angles are measured using encoders that tell you the

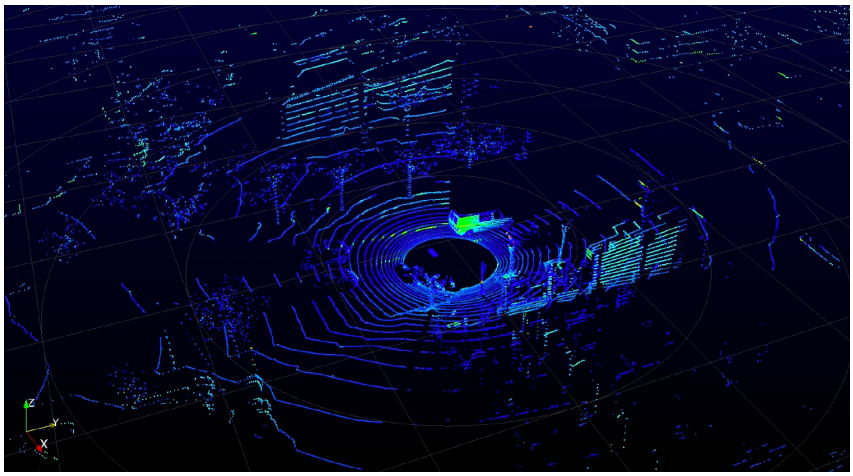


Fig. 8: Typical raw LiDAR stream.

orientation of the mirror, and the range is measured using the time of flight as we've seen before.

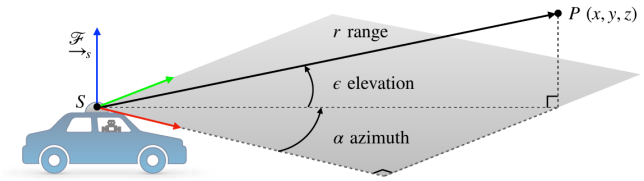


Fig. 9: Typical raw LiDAR stream.

Remark 5.2. For Velodyne type LIDARs, the elevation angle is fixed for a given scan line.

Now, suppose we want to determine the cartesian XYZ coordinates of our scanned point in the sensor frame, which is something we often want to do when we are combining multiple LiDAR scans into a map. To convert from spherical to Cartesian coordinates, we use the same formulas you would havve encountered in your mechanics classes.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \mathbf{h}^{-1}(r, \alpha, \epsilon) = \begin{bmatrix} r\cos(\alpha)\cos(\epsilon) \\ r\sin(\alpha)\cos(\epsilon) \\ \sin(\epsilon) \end{bmatrix} \tag{3}$$

This gives us an inverse sensor model. We say this is the inverse model because our actual measurements are given in spherical coordinates, and we are trying to reconstruct the Cartesian coordinates at the points that gave rise to them.

Remark 5.3. We can obviously get the forward model from the inverse model by using

$$\begin{bmatrix} r \\ \alpha \\ \epsilon \end{bmatrix} = \mathbf{h}(x, y, z) = \begin{bmatrix} \sqrt{x^2 + y^2 + z^2} \\ \arctan(\frac{x}{y}) \\ \arcsin(\frac{z}{\sqrt{x^2+y^2+z^2}} \end{bmatrix} \tag{4}$$

This is our forward sensor model for a 3D LiDAR, which given a set of Cartesian coordinates defines what the sensor would actually report.

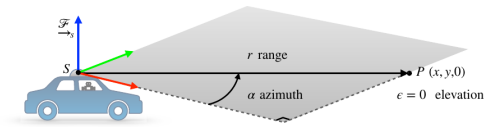
Note that we have not said anything about measurement noise yet. Most of the time the self-driving cars we're working with use 3D LIDAR sensors like the Velodyne, but sometimes you might want to use a 2D LIDAR on its own, whether for detecting obstacles or for state estimation in more structured environments such as parking garages. Some cars have multiple 2D LiDARs strategically placed to act as a single 3D LiDAR, covering different areas with a greater lesser density of measurements.

Remark 5.4. **2D LiDAR**

For 2D LIDARs, we use exactly the same forward and inverse sensor models. However, the elevation angle enhance the z component of the 3D point in the sensor frame are both zero. In other words, all of our measurements are confined to the XY plane of the sensor, and our spherical coordinates collapsed to the familiar 2D polar coordinates. Figure 10 summarizes the 2D case.

5.3 Measurement noise

We have seen now how to convert between spherical coordinates measured by the sensor and the cartesian coordinates that we will typically be interested in for state estimation, but what about measurement noise? For LiDAR sensors, there are several important sources of noise to consider. First, there is uncertainty in the exact time of arrival of the reflected signal, which comes from the fact that the stopwatch we use to compute the time of flight necessarily has a limited resolution. Similarly, there is uncertainty in the exact orientation of the mirror in 2D and 3D LIDARs since the encoder is used to measure this also have limited resolution. Another important factor is the interaction with the target surface which can degrade the return signal. For example, if the surface is completely



Inverse Sensor Model

$$\begin{bmatrix} x \\ y \\ 0 \end{bmatrix} = \mathbf{h}^{-1}(r, \alpha, 0) = \begin{bmatrix} r\cos\alpha \\ r\sin\alpha \\ 0 \end{bmatrix}$$

Forwards Sensor Model

$$\begin{bmatrix} r \\ \alpha \\ 0 \end{bmatrix} = \mathbf{h}(x, y, 0) = \begin{bmatrix} \sqrt{x^2 + y^2} \\ \tan^{-1}\left(\frac{y}{x}\right) \\ 0 \end{bmatrix}$$

Fig. 10: 2D LiDAR.

black, it might absorb most of the laser pulse. Or if it is very shiny like a mirror, the laser pulse might be scattered completely away from the original pulse direction. In both cases, the LiDAR will typically report a maximum range error, which could mean there is empty space along the beam direction, or that the pulse encountered a highly absorptive or highly reflective surface. In other words, you simply can not tell if something is there or not, and this can be a problem for safety if your self-driving car is relying on LiDAR alone to detect and avoid obstacles. Finally, the speed of light actually varies depending on the material it is traveling through. The temperature and humidity of the air can also suddenly affect the speed of light in our time-of-flight calculations for example.

These factors are commonly accounted for by assuming additive zero-mean Gaussian noise on the spherical coordinates with an empirically determined or manually tuned covariance. The Gaussian noise model is particularly convenient for state estimation even if it is not perfectly accurate in most cases. Another very important source of error that can not be accounted for so easily is motion distortion, which arises because the vehicle the LiDAR is attached to is usually moving relative to the environment it's scanning. Although the car is unlikely to be moving at an appreciable fraction of the speed of light, it is often going to be moving at an appreciable fraction of the rotation speed of the sensor itself, which is typically around 5-20 hertz when scanning objects at distances of 10 to a 100 meters. This means that every single point in a LiDAR sweep is taken from a slightly different position and a slightly different orientation, and this can cause artifacts such as duplicate objects to appear in the LIDAR scans. This makes it much harder for a self-driving car to understand its environment, and correcting this motion distortion usually requires an accurate motion model for the vehicle provided by GPS and INS for example.

5.4 Summary

To recap, LIDAR sensors measure distances by emitting pulse laser light and measuring the time of flight of the pulse. 2D or 3D LIDAR is extend this principle by using a mirror to sweep the laser across the environment and measure distances in many directions. We will look more closely at the point clouds created by 2D and 3D LIDARs, and how we can use them for state estimation on-board our self-driving car.

6 LiDAR sensor model and point clouds

In section 5, we talked about the basic operating principles of LiDAR, one of the most popular sensor choices for self-driving cars. In the next two sections we will learn how we can use the point cloud generated by LiDAR sensors to do state estimation for our self-driving car.

By the end of this section, you'll be able to describe the basic point cloud data structure used to store LiDAR scans. Describe common spatial operations on point clouds such as rotation and scaling. Use the method of least squares to fit a plane to a point cloud in order to detect the road or other surfaces.

To begin with, recall that a 3D LIDAR sensor returns measurements of range, elevation angle, and azimuth angle for every point that it scans. We know how to convert these spherical coordinates into Cartesian x, y, z coordinates using the inverse sensor model, see equation 3 so we can build up a large point cloud using all the measurements from a LiDAR scan. For some LiDAR setups, it is not uncommon for these point clouds to contain upwards of a million points or more.

So what can we do these massive point clouds? Let us consider an example of a point cloud we might encounter in the real world. Let us say our LIDAR scans a nearby tree off on the side of the road, and produces a point cloud that looks like this.

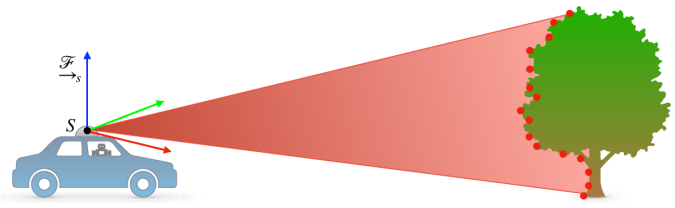


Fig. 11: Assumed LiDAR scan.

We only see points on the part of the tree that is facing us because the tree and the leaves reflect infrared light. The first question you might ask is how do we keep track of all of these points? What kinds of data structures should we use to work with them? One common solution is to assign an index to each of the points, say point 1 through point n, and store the x, y, z coordinates of each point as a 3 by 1 column vector. From there, you could think about storing each of these vectors in a list, or you could stack them side by side into a matrix that we'll call $\mathbf{P}$. Doing it this way make it easier to work with the standard linear algebra libraries, like the Python NumPy library, which lets us take advantage of fast matrix operations rather than iterating over a list and treating each vector independently. So what kind of operations are we talking about?

There are three basic spatial operations that are important for carrying out state estimation with point clouds.

- Translation
- Rotation
- Scaling

We will talk about each of these in turn. When we think about spatial operations on point clouds, our intuition might be to think in terms of physically manipulating the point cloud while our reference frame stays fixed. But for state estimation, it's more useful to think about things the other way around. Objects in the world mostly stay put while the reference frame attached to the vehicle moves and observes the world from different perspectives.

6.1 Basic spatial operations

So let's think about how translating our reference frame, say, by driving for a ten meters will affect our perception of a single point in point cloud.

We can start by drawing the vector from the origin of our sensor frame, S, to a point, P. Now, consider a second frame, $S^{'}$, whose origin has been translated relative to S due to motion of the vehicle, see Figure 12.

Note that the basis vectors of frame $S^{'}$ are the same as the basis vectors of frame S. Only the origin has moved. We can draw another vector from the origin of $S^{'}$ to the point P. Immediately, we notice the resulting vector, indicated here, is just the tip to tail sum of the other two vectors, see Figure 12. These vectors are just geometric objects until we express them in a coordinate system. What we are after are the coordinates of the point P in frame $S^{'}$. We can get these easily by just subtracting the frame-to-frame translation vector from the coordinates of P in frame S. This extends easily to a batch operation on the full point cloud by simply tiling the frame-to-frame translation in a big matrix $\mathbf{R}$, and subtracting it from the point cloud matrix.

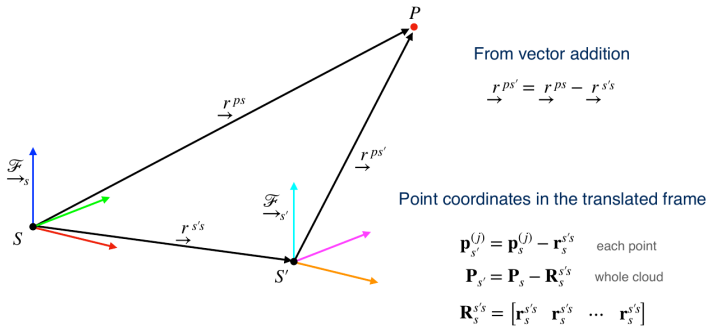


Fig. 12: Translation operation.

Remark 6.1. Depending on the language or linear algebra library you are using, you probably will nott need to build this $\mathbf{R}$ matrix explicitly. In Python, for example, the NumPy library is smart enough to repeat the frame-to-frame translation implicitly using broadcasting semantics.

Now, let's think about what happens if rotate our reference frame instead of translating it. Again, keep in mind that we're not changing the physical point P, only our view of it. So in this case, we only have to think about one vector from the origin of frame S to P. What does change in this case is actually the set of basis vectors we use to express the coordinates of the vector S to P. Remember that the rotation matrix $\mathbf{C}$ tells us how to find the coordinates of a vector in a rotated frame from the coordinates of the vector in the original frame. Thus, if we know the rotation matrix from frame S to frame $S^{'}$, all we have to do is multiply it against the coordinates of P in frame S to get the coordinates of P in frame $S^{'}$. To determine the coordinates of the entire rotated point cloud, the operation is exactly the same, thanks to the properties of matrix multiplication.

$$\mathbf{P}_{S'} = \mathbf{C}_{S'S}\mathbf{P}_S \tag{5}$$

The last spatial operation to think about is scaling, which works very similarly to rotation. But instead of changing the direction of the basis vectors in our coordinate system, we're changing their lengths. Mathematically, this just means pre-multiplying the coordinates of each point by a diagonal matrix $\mathbf{S}$ whose non-zero elements are simply the desired scaling factors along each dimension. Often but not always these scaling factors are the same, and the matrix multiplication is equivalent to multiplying by a scalar. In these cases, we say that the scaling is isotropic or equal in every direction. We can use the same matrix multiplication for individual points or for the entire point cloud, just like we did

for rotations.

Usually, the transformations we're interested in are a combination of translation and rotation and sometimes scaling. For example, we are often interested in estimating the translation and rotation that best aligns to point clouds so that we can estimate the motion of our self-driving car. Fortunately for us, it's easy to combine all three operations into a single equation By first translating each vector, then rotating into the new frame, and finally applying any scaling. Of course, this operation extends to the batch case as well. So we have seen how to apply basic spatial operations to point clouds.

6.2 Plane fitting

One of the most common and important applications of plane-fitting for self-driving cars is figuring out where the road surface is and predicting where it is going to be as the car continues driving. If you think back to your high school geometry classes, you might remember the equation of a plane in 3D.

$$z = a + bx + cy \tag{6}$$

This equation tells you how the height of the plane z changes as you move around in the x and y directions. It depends on three parameters, a, b, and c, which tells you the slope of the plane in each direction and where the z axis intersects the plane. So in our case, we have a bunch of measurements of x, y and z from our LIDAR point cloud, and we want to find values for the parameters a, b, and c that give us the plane of best fit through these points. To do this, we are going to use least-squares estimation. We will start by defining a measurement error e for each point in the point cloud. e is just going to be the difference between the predicted value of our dependent variable $\hat{z}$ and the actual observed value of z.

$$e_j = \hat{z}_j - z_j = \hat{a} + \hat{b}x + \hat{c}y - z_j, j = 1, \ldots, n \tag{7}$$

We get $\hat{z}$ simply by plugging our current guess for the parameters $\hat{a}$, $\hat{b}$, and $\hat{c}$, and the actual values of x and y in. In this case, the error, e, that we are considering, is for a bumpy road surface, for example. Note that for the moment, we are ignoring the actual errors in the LiDAR measurements themselves, which also have an effect.

We can stack all of these error terms into matrix form so we have a big matrix of coefficients called $\mathbf{A}$. Multiplied by our parameter vector $\mathbf{x}$, minus our stack measurements $\mathbf{b}$.

$$\underbrace{\begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{bmatrix}}_{\mathbf{e}} = \underbrace{\begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ \vdots & \vdots & \vdots \\ 1 & x_n & y_n \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} a \\ b \\ c \end{bmatrix}}_{\mathbf{x}} - \underbrace{\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix}}_{\mathbf{b}} \tag{8}$$

You can work out the matrix multiplication yourself to see that we get back the same measurement error equations we started out with.

Now, all we have to do is minimize the square of this error and we will have our solution. We can start by multiplying out the square to get a matrix polynomial in the parameter vector $\mathbf{x}$. From there, we take the partial derivative of the squared error function with respect to the parameter vector $\mathbf{x}$ and set it to 0 to find the minimum. This gives us the linear system we will need to solve to obtain the final least squares estimate.

$$\hat{\mathbf{x}} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b} \tag{9}$$

We can solve this linear system using an efficient numerical solver like Python NumPy's solve function. Or just use the pseudo inverse to get our final answer for the plane parameters.

One important thing to notice here is that we did not account for sensor noise in our x, y, z measurements. All we did was to find the plane of best fit through a set of points. It is certainly possible to set this problem up in a more sophisticated way that does account for sensor noise. You could use a batch approach similar to what we just discussed, or you could even think about including the road parameters in the column filter to estimate them on the fly as the sensor data comes in.

The best solution for your self-driving application will depend on how much you trust your LiDAR data and how much thought you want to give to uncertainty in the road surface. Now, although all of the operations we've described here can be easily implemented with NumPy or any other linear algebra library, there is a fantastic open source tool called the Point Cloud Library, or PCL, that provides all sorts of useful functions for working with point clouds. In fact, it is so useful that you'll find it everywhere in industry. The core library is built with C++, but there are unofficial Python bindings available as well. If you want to learn more about the features PCL.

6.3 Summary

In summary, we have seen that point clouds are a way of capturing all of the measurements from a LiDAR scan. They are often stored as a big matrix. We

saw how we can use linear algebra to do useful operations on point clouds, like translating, rotating, and scaling. We also saw how we can use the least squares algorithm to fit a 3D plane to a point cloud to find the road surface. The Point Cloud Library, or PCL, implements a bunch of useful tools for working with point clouds in C++. One of the most useful algorithms in PCL is called the iterative closest point algorithm, or ICP, which is a common method for estimating the motion of a self-driving car using two LIDAR point clouds.

7 Pose estimation from LiDAR data

In this section, we will talk about how we can actually use these operations with real point clouds to estimate the motion of a self-driving car. The way that we do this in general is by solving something called the point set registration problem, which is one of the most important problems in computer vision and pattern recognition.

By the end of this section, you will be able to describe the point set registration problem and how it can be used for state estimation, describe and implement the Iterative Closest Point or ICP algorithm for point set registration, and understand some common pitfalls of using ICP for state estimation on self-driving cars.

7.1 The point set registration problem

Let us explore this problem by returning to our example of a self-driving car observing a tree from a LIDAR mounted on the cars roof. At time $t = 1$ say, the LiDAR returns a point cloud that follows the contour of the tree as before. The coordinates of every point in the point cloud are given relative to the pose of the lidar at the time of the scan, we will call this coordinate frame S. At time $t = 2$ the car is driven a bit further ahead, but the LiDAR can still see the tree and return a second point cloud whose coordinates are again specified relative to the pose of the lidar at time t equals two, we will call this coordinate frame $S^{'}$. Now, the point set registration problems says, given two point clouds in two different coordinate frames, and with the knowledge that they correspond to or contain the same object in the world, how shall we align them to determine how the sensor must have moved between the two scans? More specifically, we want to figure out the optimal translation and the optimal rotation between the two sensor reference frames that minimizes the distance between the 2 point clouds. To keep things simple, we are going to pretend that we have an ideal LiDAR sensor that measures the entire point cloud all at once, so that we can ignore the effects of motion distortion. Now, if we somehow knew that each and every point in the second point cloud corresponded to a specific point in the first point cloud, and we knew ahead of time which points corresponding to

which, we could solve this problem easily. All we would have to do is find the translation and rotation that lines up each point with its twin.

In this example, our ideal rotation matrix would be the identity matrix, that is no rotation at all, and a ideal translation would be along the cars forward direction. The problem is that, in general we do not know which points correspond to each other. Feature matching which is one way of determining correspondences between points using camera data. For now let us think about how we might solve this problem without knowing any of the correspondences ahead of time. The most popular algorithm for solving this kind of problem is called the Iterative Closest Point algorithm or ICP for short.

7.2 Iterative closest point

The basic intuition behind ICP is this, when we find the optimal translation and rotation, or if we know something about them in advance, and we use them to transform one point cloud into the coordinate frame of the other, the pairs of points that truly correspond to each other will be the ones that are closest to each other in a Euclidean sense. This makes sense if you consider the simplified case where our LiDAR sensors scans exactly the same points just from two different vantage points. In this case, there is a one-to-one mapping between points in the two scans, and this mapping is completely determined by the motion of the sensor.

This is the ideal case for when we found the best possible translation and rotation, but what if we do not have the optimal translation and rotation, how do we decide which points actually go together? Well, with ICP we use a simple heuristic, and say that for each point in one cloud, the best candidate for a corresponding point in the other cloud is the point that is closest to it right now. The idea here is that this heuristic should give us the correspondences that let us make our next guess for the translation and rotation, that is a little bit better than our current guess. As our guesses get better and better, our correspondences should also get better and better until we eventually converge to the optimal motion and the optimal correspondences.

This iterative optimization scheme using the closest point heuristic is where ICP gets its name. It is important to note that the word closest implies that we will find points that lie close together after applying the frame to frame transformation only. This is, because the vehicle is moving it is almost never the case that a laser beam will hit exactly the same surface point twice.

Let us now talk about the steps in the ICP algorithm. First and most important is that we need an initial guess for the transformation.

$$\check{\mathbf{C}}_{S'S}, \check{\mathbf{r}}_{S}^{S'S} \tag{10}$$

We need this because there is no guarantee that the closest point heuristic will actually converge to the best solution. It's very easy to get stuck in a local minimum in these kinds of iterative optimization schemes. So, we will need a good starting guess.

Now, the initial guess can come from a number of sources. One of the most common sources for robotics applications like self-driving is a motion model, which could be supplied by an IMU or by wheel odometry or something really simple like a constant velocity or even a zero velocity model. How complex the motion model needs to be to give us a good initial guess really depends on how smoothly the car is driving. If the car is moving slowly relative to the scanning rate of the LIDAR sensor, one may even use the last known pose as the initial guess. For our example, let's say we use a very noisy IMU and emotion model to provide the initial guess. The model tells us that the sensor translated forward a bit and also pitched upwards. In step two, we will use this initial guess to transform the coordinates of the points in one cloud into the reference frame of the other, and then match each point in the second cloud to the closest point in the first cloud.

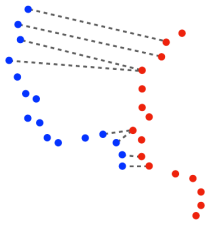


Fig. 13: Associate each point in $\mathbf{P}_{S'}$ with the nearest point in $\mathbf{P}_S$.

Figure 13 shows the associations for the first four points and the last four points. Note that there's nothing stopping two points in one cloud from being associated with the same point in the other cloud.

Step three is to take all of those match points and find the transformation that minimizes the sum of squared distances between the corresponding points. You can visualize it as wrapping an elastic band around each pair of matched points, and then letting go and waiting for the system to find its lowest energy configuration. The translation and rotation associated with this minimum energy configuration are the optimal solutions. Then, we repeat the process using the translation and rotation we just solved for as our new initial guess. We keep going until the optimization converges. Thus, in the next iteration, we use the new translation and rotation to transform the point cloud and then find the closest matches, solve for the optimal transformation, and keep going until we reach the optimum after a few iterations.

How do we actually solve for the optimal transformation in step three? One option is to use least-squares. Our goal here is to find the rotation and translation

that best aligns the two point clouds. Specifically, you want to minimize the sum of squared euclidean distances between each pair of matched points, which is one of the loss function that we've defined here.

$$\check{\mathbf{C}}_{S'S}, \check{\mathbf{r}}_S^{S'S} = argmin_{\mathbf{C}_{S'S}, \mathbf{r}_S^{S'S}} L_{LS}(\mathbf{C}_{S'S}, \mathbf{r}_S^{S'S}) \tag{11}$$

This least squares problem is a little bit more complex. This is because the rotation matrix is inside the loss function. It turns out the rotation matrices do not behave like vectors. If you add two vectors together, you get another vector. But if you add two rotation matrices together, the result is not necessarily a valid rotation matrix. In reality, 3D rotations belong to something called the special orthogonal group or SO3.

It turns out that there is a nice closed form solution to this least-squares problem which was discovered in the 1960s. We will not go through the derivation here, but the good news is that there is a simple four step algorithm you can follow to solve this problem. The first two steps are easy. First, we compute the centroid of each point cloud by taking the average over the coordinates of each point. This is exactly like calculating the center of mass in physics.

$$\boldsymbol{\mu}_S = \frac{1}{n}\sum_{j=1}^{n} \mathbf{p}_S^j, \quad \boldsymbol{\mu}_{S'} = \frac{1}{n}\sum_{j=1}^{n} \mathbf{p}_{S'}^j \tag{12}$$

Second, we work at a three-by-three matrix capturing the spread of the two point clouds.

$$\mathbf{W}_{S'S} = \frac{1}{n}\sum_{j=1}^{n} (\mathbf{p}_S^j - \boldsymbol{\mu}_S)(\mathbf{p}_{S'}^j - \boldsymbol{\mu}_{S'})^T \tag{13}$$

You can think of this $\mathbf{W}$ matrix as something like an inertia matrix you might encounter in mechanics. The $\mathbf{W}$ matrix is the quantity we are going to use to estimate the optimal rotation matrix.

Step three is actually finding the optimal rotation matrix. This step is the most complex, and it involves taking something called the singular value decomposition or SVD of the $\mathbf{W}$ matrix.

Remark 7.1. **Singular Value Decomposition or SVD**

The SVD is a way of factorizing a matrix into the product of two unitary matrices, $\mathbf{U}$ and $\mathbf{V}$, and a diagonal matrix $\mathbf{D}$, whose non-zero entries are called the singular values of the original matrix.

$$\mathbf{W}_{S'S} = \mathbf{U}\mathbf{D}\mathbf{V}^T \tag{14}$$

There are several ways to interpret the SVD. For us, it is easiest to think about $\mathbf{U}$ and $\mathbf{V}$ as rotations and the $\mathbf{D}$ matrix as a scaling matrix. Since we are dealing with rigid body motion in this problem, we do not want any scaling in a rotation estimate, so we will replace the $\mathbf{D}$ matrix with something like the identity matrix to remove the scaling.

$$\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \det\mathbf{U}\det\mathbf{V} \end{bmatrix} \tag{15}$$

It turns out that in some cases, the SVD will give us rotation matrices that are not proper rotations. That is, they also have a reflection about the axis of rotation. To make sure that we get a proper rotation, we choose the bottom right term to be the product of the determinants of $\mathbf{U}$ and $\mathbf{V}$. This number will be either plus one or minus one and will cancel out any reflection. Once we have this, all we have to do is multiply out the three matrices. This gives us back the optimal rotation matrix.

Now, that we have our rotation estimate, the last step is to recover the translation.

$$\hat{\mathbf{r}}_S^{S'S} = \boldsymbol{\mu}_S - \check{\mathbf{C}}_{S'S}^T \boldsymbol{\mu}_{S'}, \quad \check{\mathbf{C}}_{S'S} = \mathbf{U}\mathbf{D}\mathbf{V}^T \tag{16}$$

This part is very straightforward and only involves rotating the centroid of one point cloud into the reference frame of the other and then taking the difference of the coordinate vectors to find the translation that will best align the two point clouds.

One other important thing to think about, both from a safety standpoint and when we start talking about sensor fusion, is how confident we should be in the output of the ICP algorithm. There are a number of different ways to estimate the uncertainty or the covariance of the estimated motion parameters. An accurate and relatively simple method is to use this equation here.

$$\text{cov}(\hat{\mathbf{x}}) \approx \left[(\tfrac{\partial^2 L}{\partial \mathbf{x}^2})^{-1}, \tfrac{\partial^2 L}{\partial \mathbf{z} \partial \mathbf{x}}, \tfrac{\partial^2 L}{\partial \mathbf{z} \partial \mathbf{x}} (\tfrac{\partial^2 L}{\partial \mathbf{x}^2})^{-1} \right]_{\mathbf{x}=\hat{\mathbf{x}}} \tag{17}$$

This expression tells us how the covariance of the estimated motion parameters is related to the covariance of the measurements in the two point clouds using certain second-order derivatives of the least squares cost function. While its expression is accurate and fast to compute, it's a little tricky to derive these second-order derivatives when there's a constraint quantity like a rotation matrix in the mix. For our purposes, we're generally going to hand tune the ICP covariance matrix to give us acceptable results.

7.2.1 ICP variants

You have now seen the basic vanilla algorithm for the iterative closest point method, but it's not the only way of solving a problem. In fact, this algorithm is just one variant of ICP called Point-to-point ICP, which derives its name from the fact that our objective function or loss function minimizes the distance between pairs of corresponding points.

Another popular variant that works well in unstructured environments like cities or indoors is called Point-to-plain ICP. Instead of minimizing the distance between pairs of points, we fit a number of planes to the first point cloud and then minimize the distance between each point in the second cloud and its closest plane in the first cloud. These planes could represent things like walls or road surfaces. The challenging part of the algorithm is actually figuring out where all the planes are. You can check out the documentation for Point-to-plain ICP in the point cloud library for more information.

7.3 Objects in motion

Now, up until this point, we've been assuming that the objects seen by our LiDAR are stationary. What if they're moving? A common example of this is if our car is driving in traffic down a busy highway and scanning the vehicle in front of it. Both vehicles are traveling at the same speed while our self-driving car is happily collecting LiDAR data points. We ask ourselves again, what motion of the car best aligns the two point clouds? The answer we get is that we haven't moved at all. But, of course, we did move, just not relative to the vehicle directly ahead of us. This is obviously a contrived example. In reality, the point cloud would also include many stationary objects like roads, and buildings, and trees. But naively using ICP in the presence of moving objects will tend to pull our motion estimates away from the true motion. So we need to be careful to exclude or mitigate the effects of outlying points that violate our assumptions of a stationary world. One way to do this is by fusing ICP motion estimates with GPS and INS estimates. Another option is to identify and ignore moving objects, which we could do with some of the computer vision. An even simpler approach for dealing with outliers like these is to choose a different loss function that is less sensitive to large errors induced by outliers than our standard squared error loss. The class of loss functions that have this property are called **Robust Loss Functions** or robust cost functions, and there are several to choose from. We can write this out mathematically by generalizing our least squares loss function so that the contribution of each error is not simply the square of its magnitude, but rather, some other function ρ. Some popular choices for robust loss functions include the Absolute error or L1 norm, the Cauchy loss, and the Huber loss are shown in Figure 14.

The key difference is that the slope of the loss function does not continue to increase as the errors become larger, but rather, it remains constant or it tapers

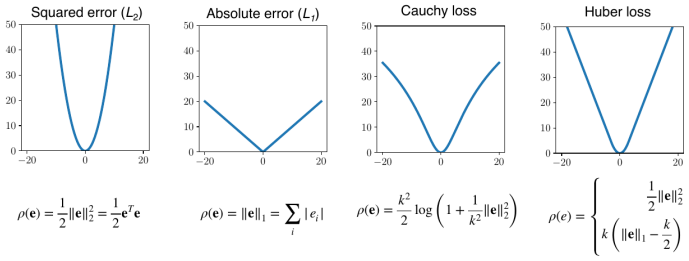


Fig. 14: Robust Loss Functions.

off. Robust loss functions make the ICP problems slightly more difficult because we can no longer derive a nice closed form solution for the point cloud alignment step, and this means we need to add another iterative optimization step inside our main loop. However, the benefits cannot weigh the added complexity.

7.4 Summary

In summary, the Iterative Closest Point or ICP algorithm is a way to determine the motion of a self-driving car by aligning point clouds from LiDAR or other sensors. ICP works by iteratively minimizing the Euclidean distance between neighboring points in each point cloud which is where the algorithm gets its name. Moving objects can be a problem for ICP since they violate the stationary world assumption that ICP is based on. Outlier measurements like these can be mitigated with a number of techniques including Robust Loss Functions, which assign less weight to large errors than the usual squared error loss.

8 Answers to Questions

Answer to questions in section 2

1. What are the differences between exteroceptive sensors and proprioceptive sensors? (Select all that apply)
2. Which of the following exteroceptive sensors would you use in harsh sunlight?
3. Why is synchronization and timing accuracy important in the self driving system? Choose the primary reason.
4. Your autonomous vehicle is driving on the German autobahn at $150km/h$ and you wish to maintain safe following distances with other vehicles. Assuming a safe following distance of $2s$, what is the distance (in m) required between vehicles? Round your answer to 2 decimal places.
5. Using the same speed of $150km/h$, what is the braking distance (in m) required for emergency stops? Assume an aggressive deceleration of $5m/s^2$. Round your answer to two decimal places.
6. Suppose your vehicle was using long range cameras for sensing forward distance, but it is now nighttime and the images captured are too dark. Which of the following sensors can be used to compensate?
7. What are the differences between an occupancy grid and a localization map? (Select all that apply)
8. The vehicle steps through the software architecture and arrives at the controller stage. What information is required for the controller to output its commands to the vehicle?
9. What is (are) the role(s) of the system supervisor? (Select all that apply)
10. Which of the following tasks should be assigned to the local planner?
11. What common objects in the environment appear in the occupancy grid?
12. Which of the following maps contain roadway speed limits?

References

[1] SAE *Taxonomy of Driving* $https://www.sae.org/standards/content/j3016\_201806/?PC = DL2BUY$

[2] SAE *SAE J3016 Taxonomy and Definitions Document* $https://drive.google.com/open?id = 1xtOqFVJvOElXjXqf4RAwXZkI\_EwbxFMg$

[3] Åström K. J., Murray R. M. *Feedback Systems. An Introduction for Scientists and Engineers*

[4] Philip , Florent Altche1, Brigitte dAndrea-Novel, and Arnaud de La Fortelle *The Kinematic Bicycle Model: a Consistent Model for Planning Feasible Trajectories for Autonomous Vehicles?* HAL Id: hal-01520869, https://hal-polytechnique.archives-ouvertes.fr/hal-01520869

[5] Marcos R. O., A. Maximo *Model Predictive Controller for Trajectory Tracking by Differential Drive Robot with Actuation constraints*