Chapter 4

Rigid-body transforms and mapping

Reading

1. LaValle Chapter 3.2 for rotation matrices, Chapter 4.1-4.2 for quaternions
2. Thrun Chapter 9.1-9.2 for occupancy grids
3. OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees http://www.arminhornung.de/Research/pub/hornung13auro.pdf, also see https://octomap.github.io.
4. Robot Operating System http://www.willowgarage.com/sites/default/files/icraoss09-ROS.pdf, Optional: Lightweight Communications and Marshalling (LCM) system https://people.csail.mit.edu/albert/pubs/2010-huang-olson-moore-lcm-iros.pdf
5. A Perception-Driven Autonomous Urban Vehicle https://april.eecs.umich.edu/media/pdfs/mitduc2009.pdf
6. Optional reading: Thrun Chapter 10 for simultaneous localization and mapping

In the previous chapter, we looked at ways to estimate the state of the robot in the physical world. We kept our formulation abstract, e.g., the way the robot moves was captured by an abstract expression like $x_{k+1} = f(x_k, u_k) + \epsilon$ and observations $y_k = g(x_k) + \nu$ were similarly opaque. In other to actually implement state estimation algorithms on real robots, we need to put concrete functions in place of f, g.

This is easy to do for some robots, e.g., the robot in Problem 1 in Homework 1 moved across cells. Of course real robots are a bit more complicated, e.g., a car cannot move sideways (which is a huge headache when you parallel park). In the first half of this chapter, we will look at how to model the dynamics f using rigid-body transforms.

The story of measurement models and sensors is similar. Although we need to write explicit formulae in place of the abstract function g. In the second half, we will study occupancy grids and dig deeper into a typical state-estimation problem in robotics, namely that of mapping the location of objects in the world around the robot.

4.1 Rigid-Body Transformations

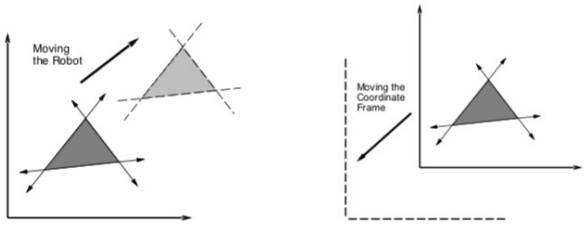
Let us imagine that the robot has a rigid body, we think of this as a subset $A \subset \mathbb{R}^2$. Say the robot is a disc

$$A = \left\{(x, y) \in \mathbb{R}^2 : x^2 + y^2 \leq 1\right\}.$$

This set A changes as the robot moves around, e.g., if the center of mass of the robot is translated by $x_t, y_t \in \mathbb{R}$ the set A changes to

$$A' = \{(x + x_t, y + y_t) : (x, y) \in A\}.$$

The concept of "degrees of freedom" denotes the maximum number of independent parameters needed to completely characterize the transformation applied to a robot. Since the set of allowed values (x_t, y_t) is a two-dimensional subset of $\mathbb{R}^2$, then the degrees of freedom available to a translating robot is two.



(a) Translation of the robot (b) Translation of the frame

As the above figure shows, there are two ways of thinking about this transformation. We can either think of the robot transforming while the co-ordinate frame of the world is fixed, or we can think of it as the robot remaining stationary and the co-ordinate frame undergoing a translation. The second style is useful if you want to imagine things from the robot's perspective. But the first one feels much more natural and we will therefore exclusively use the first notion.

If the same robot if it where rotated counterclockwise by some angle $\theta \in [0, 2\pi]$, we would map

$$(x, y) \mapsto (x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta).$$

Such a map can be written as multiplication by a 2×2 rotation matrix

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}. \tag{4.1}$$

to get

$$\begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix} = R(\theta)\begin{bmatrix} x \\ y \end{bmatrix}.$$

The transformed robot is thus given by

$$A' = \left\{ R\begin{bmatrix} x \\ y \end{bmatrix} : (x,y) \in A \right\}.$$

If we perform both rotation and translation, we can the transformation using a single matrix

$$T = \begin{bmatrix} \cos\theta & -\sin\theta & x_t \\ \sin\theta & \cos\theta & y_t \\ 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

and this transformation looks like

$$\begin{bmatrix} x\cos\theta - y\sin\theta + x_t \\ x\sin\theta + y\cos\theta + y_t \\ 1 \end{bmatrix} = T\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.$$

The point $(x, y, 1) \in \mathbb{R}^3$ is called homogeneous coordinate space corresponding to $(x, y) \in \mathbb{R}^3$ and the matrix T is called a *homogeneous transformation matrix*. The peculiar names comes from the fact that even if the matrix T maps rotations and translations of rigid bodies $A \subset \mathbb{R}^2$, it is just a linear transformation of the point $(x, y, 1)$ if viewed in the larger space $\mathbb{R}^3$.

ℹ It is important to remember that T represents rotation *followed* by a translation, not the other way around.

Rigid-body transformations The transformations $R \in \mathbb{R}^{2\times2}$ or $T \in \mathbb{R}^{3\times3}$ are called rigid-body transformations. Mathematically, it means that they do not cause the distance between any two points inside the set A to change. Rigid-body transformations are what are called an orthogonal group in mathematics.

A group is a mathematical object which imposes certain conditions upon how two operations, e.g., rotations, can be composed together. For instance, if G is the group of rotations, then (i) the composition of two rotations is a rotation, we say that it satisfies closure $R(\theta_1)R(\theta_2) \in G$, (ii) rotations are associative

$$R(\theta_1)\left\{R(\theta_2)R(\theta_3)\right\} = \left\{R(\theta_1)R(\theta_2)\right\}R(\theta_3),$$

and, (iii) there exists an identity and inverse rotation

$$R(0), R(-\theta) \in G.$$

An orthogonal group is a group whose operations preserve distances in Euclidean space, i.e., $g \in G$ is an element of the group that acts on two points

$x, y \in \mathbb{R}^d$ then

$$\|g(x) - g(y)\| = \|x - y\|.$$

If we identify the basis in Euclidean space to be the set of orthonormal vectors $\{e_1, \ldots, e_d\}$, then equivalently, the orthogonal group $O(d)$ is the set of orthogonal matrices

$$O(d) := \left\{O \in \mathbb{R}^{d\times d} : OO^\top = O^\top O = I\right\}.$$

This implies that the square of the determinant of any element $a \in O(d)$ is 1, i.e., $\det(a) = \pm 1$.

Check that any rotation matrix R belongs to an orthogonal group.

The Special Orthogonal Group is a sub-group of the orthogonal group where the determinant of each element is +1. You can see that rotations are a special orthogonal group. We denote rotations of objects in $\mathbb{R}^2$ as

$$SO(2) := \left\{R \in \mathbb{R}^{2\times 2} : R^\top R = RR^\top = I, \det(R) = 1\right\}. \tag{4.3}$$

Each group element $g \in SO(2)$ denotes a rotation of the XY-plane about the Z-axis. The group of 3D rotations is called the Special Orthogonal Group $SO(3)$ and is defined similarly

$$SO(3) := \left\{R \in \mathbb{R}^{3\times 3} : R^\top R = RR^\top = I, \det(R) = 1\right\}. \tag{4.4}$$

The Special Euclidean Group $SE(2)$ is simply a composition of a 2D rotation $R \in SO(2)$ and a 2D translation $\mathbb{R}^2 \ni v \equiv (x_t, y_t)$

$$SE(2) = \left\{\begin{bmatrix} R & v \\ 0 & 1 \end{bmatrix} : R \in SO(2), v \in \mathbb{R}^2\right\} \subset \mathbb{R}^{3\times 3}. \tag{4.5}$$

The Special Euclidean Group $SE(3)$ is defined similarly as

$$SE(3) = \left\{\begin{bmatrix} R & v \\ 0 & 1 \end{bmatrix} : R \in SO(3), v \in \mathbb{R}^3\right\} \subset \mathbb{R}^{4\times 4}; \tag{4.6}$$

again, remember that it is rotation *followed* by a translation.

4.1.1 3D transformations

Translations and rotations in 3D are conceptually similar to the two-dimensional case; however the details appear a bit more difficult because rotations in 3D are more complicated.

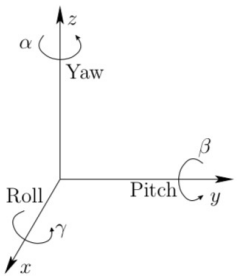


Figure 4.1: Any three-dimensional rotation can be described as a sequence of rotations about each of the cardinal axes. We usually give these specific names: rotation about the Z-axis is called yaw, rotation about the X-axis is called roll and rotation about Y-axis is called pitch. You should commit this picture and these names to memory because it will be of enormous to think about these rotations intuitively.

Euler angles We know that a pure counter-clockwise rotation about one of the axes is written in terms of a matrix, say yaw of α-radians about the Z-axis

$$R_z(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Notice that this is a 3×3 matrix that keeps the Z-coordinate unchanged and only affects the other two coordinates. Similarly we have for pitch (β about the Y-axis) and roll (γ about the X-axis)

$$R_y(\beta) = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix}, R_x(\gamma) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{bmatrix}.$$

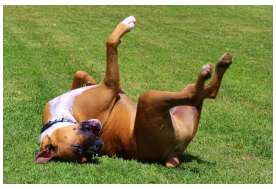
By convention, a rotation matrix in three dimensions is understood as a sequential application of rotations, *first roll, then pitch, and then yaw*

$$\mathbb{R}^{3\times3} = R(\alpha,\beta,\gamma) = R_z(\alpha)R_y(\beta)R_x(\gamma). \tag{4.7}$$

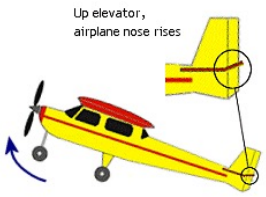
The angles (α,γ,γ) are called Euler angles. Imagine how the body frame of the robot changes as successive rotations are applied. If you were sitting in a car, a pure yaw would be similar to the car turning left; the Z-axis corresponding to this yaw would however only be pointing straight up perpendicular to the ground if you had not performed a roll/pitch before. If you had, the Z-axis of the body frame with respect to the world will be tiled.

Order of 3D rotations matter

ℹ Here is how I remember these names. Say you are driving a car, usually in robotics we take the X-axis to be longitudinally forward, the Y-axis is your left hand if you are in the driver's seat and the Z-axis points up by the right-hand thumb rule. Roll



is what a dog does when it rolls, it rotates about the X-axis. Pitch is what a plane



does when it takes off, its nose lifts up and it rotates about the Y-axis. Yaw is the one leftover.

$$R_A(\alpha,\beta,\gamma) = R_3(\gamma)R_2(\beta)R_1(\alpha)$$

$$\begin{bmatrix} \cos\gamma\cos\beta & -\sin\gamma\cos\alpha+\cos\gamma\sin\beta\sin\alpha & \sin\gamma\sin\alpha+\cos\gamma\sin\beta\cos\alpha \\ \sin\gamma\cos\beta & \cos\gamma\cos\alpha+\sin\gamma\sin\beta\sin\alpha & -\cos\gamma\sin\alpha+\sin\gamma\sin\beta\cos\alpha \\ -\sin\beta & \cos\beta\sin\alpha & \cos\beta\cos\alpha \end{bmatrix}$$

$$R_B(\alpha,\beta,\gamma) = R_1(\alpha)R_2(\beta)R_3(\gamma)$$

$$\begin{bmatrix} \cos\gamma\cos\beta & -\sin\gamma\cos\beta & \sin\beta \\ \cos\gamma\sin\beta\sin\alpha+\sin\gamma\cos\alpha & -\sin\gamma\sin\beta\sin\alpha+\cos\gamma\cos\alpha & -\cos\beta\sin\alpha \\ -\cos\gamma\sin\beta\cos\alpha+\sin\gamma\sin\alpha & \sin\gamma\sin\beta\cos\alpha+\cos\gamma\sin\alpha & \cos\beta\cos\alpha \end{bmatrix}$$

Another important thing to note is that one single parameter determines all possible rotations about one axis, i.e., $SO(2)$. But three Euler angles are used to parameterize general rotations in three-dimensions.

Rotation matrices to Euler angles We can back-calculate the Euler angles from a rotation matrix as follows. Given an arbitrary matrix

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix},$$

we set

$$\begin{aligned} \alpha &= \tan^{-1}(r_{21}/r_{11}) \\ \beta &= \tan^{-1}\left(-r_{31}/\sqrt{r_{32}^2+r_{33}^2}\right) \\ \gamma &= \tan^{-1}(r_{32}/r_{33}). \end{aligned} \tag{4.8}$$

For each angle, the corresponding quadrant for the Euler angle is determined using the signs of the numerator and the denominator. So you should use the function atan2 in Python/C++ to implement these expressions correctly. Notice that some of the expressions have r_{11} and r_{33} in the denominator, this means that we need $r_{11} = \cos\alpha\cos\beta = 0$ and $r_{33} = \cos\beta\cos\gamma = 0$. Euler angles do not completely parametrize the group of rotations $SO(3)$. This is really an indicator that this particular physical rotation can be parameterized in two different ways using Euler angles, so the map from rotation matrices to Euler angles is not unique.

Homogeneous coordinates in three dimensions Just like the 2D case, we can define a 4×4 matrix that transforms points $(x, y, z) \in \mathbb{R}^3$ to their new locations after a rotation by Euler angles (α, β, γ) and a translation by a vector $v = (x_t, y_t, z_t) \in \mathbb{R}^3$

$$T = \begin{bmatrix} R(\alpha,\beta,\gamma) & v \\ 0 & 1 \end{bmatrix}.$$

4.1.2 Rodrigues' formula: an alternate view of rotations

Consider a point $r(t) \in \mathbb{R}^3$ that is being rotated about an axis denoted by a unit vector $\omega \in \mathbb{R}^3$ with an angular velocity of 1 radian/sec. The instantaneous

linear velocity of the head of the vector is

$$\dot{r}(t) = \omega \times r(t) \equiv \hat{w} r(t) \tag{4.9}$$

where the $\times$ denotes the cross-product of the two vectors $a, b \in \mathbb{R}^3$

$$a \times b = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix}$$

which we can equivalently denote as a matrix vector multiplication $a \times b = \hat{a} b$ where

$$\hat{a} = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix}. \tag{4.10}$$

is a skew-symmetric matrix. The solution of the differential equation (4.9) at time $t = \theta$ is

$$r(\theta) = \exp(\hat{\omega}\theta)\, r(0)$$

where the matrix exponential of a matrix A is defined as

$$\exp(A) = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \ldots = \sum_{k=0}^{\infty} \frac{A^k}{k!}.$$

This is an interesting observation: a rotation about a fixed axis ω by an angle θ can represented by the matrix

$$R = \exp(\hat{w}\, \theta).$$

You can check that this matrix is indeed a rotation by showing that $R^\top R = I$ and that $\det(R) = +1$. We can expand the matrix exponential and collect odd and even powers of $\hat{\omega}$ to get

$$R = I + \sin\theta\, \hat{\omega} + (1 - \cos\theta)\hat{w}^2. \tag{4.11}$$

which is the Rodrigues' formula that relates the angle θ and the axis ω to the rotation matrix. We can also go in the opposite direction, i.e., given a matrix R calculate what angle θ and axis ω it corresponds to using

$$\begin{aligned} \cos\theta &= \frac{\text{tr}(R) - 1}{2} \\ \hat{w} &= \frac{R - R^\top}{2\sin\theta}. \end{aligned} \tag{4.12}$$

4.2 Quaternions

We know two ways to think about rotations: we can either think in terms of the three Euler angles (α, β, γ), or we can consider a rotation matrix $R \in \mathbb{R}^{3\times 3}$. We also know ways to go to and fro between these two forms with the caveat that solving for Euler angles using (4.8) may be degenerate in some cases. While rotation matrices are the most general representation of rotations, using

them in computer code is cumbersome (it is, after all, a matrix of 9 elements), imagine an EKF where the state is a rotation matrix. You can certainly implement the same filter using Euler angles but doing so requires special care due to the degeneracies. Quaternions are a neat way to avoid both these problems, they parametrize the space of rotations using 4 numbers and do not have degeneracies.

The central idea behind quaternions is Euler's theorem which says that any 3D rotation can be considered as a pure rotation by an angle $\theta \in \mathbb{R}$ about an axis given by the unit vector ω. This is the result that we also exploited in Rodrigues' formula.

ℹ Quaternions were invented by British mathematician William Rowan Hamilton while walking along a bridge with his wife. He was quite excited by this discovery and promptly graffitied the expression into the stone of the bridge

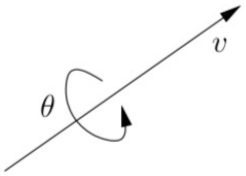


Figure 4.2: Any rotation in 3D can be represented using a unit vector ω and an angle $\theta \in \mathbb{R}$. Notice that there are two ways to encode the same rotation, the unit vector $-\omega$ and angle $2\pi - \theta$ would give the same rotation. Mathematicians say this as quaternions being a double-cover of $SO(3)$.

A quaternion q as a four-dimensional vector $q \equiv (u_0, u_1, u_2, u_3)$ and we write it as

$$\begin{aligned} &q \equiv (u_0, u), \text{ or} \\ &q = u_0 + u_1 i + u_2 j + u_3 k, \end{aligned} \tag{4.13}$$

with i, j, k being three "imaginary" components of the quaternion with "complex-numbers like" relationships

$$i^2 = j^2 = k^2 = ijk = -1. \tag{4.14}$$

It follows from these relationships that

$$ij = -ji = k, ki = -ik = j, \text{ and } jk = -jk = i.$$

Although you may be tempted to think about this, these imaginary components i, j, k have no relationship with the square roots of negative unity used to define standard complex numbers. You should simply think of the quaternion as a four-dimensional vector in some space. A unit quaternion, i.e., one with

$$u_0^2 + u_1^2 + u_2^2 + u_3^2 = 1,$$

is special: unit quaternions can be used to represent rotations in 3D.

Quaternion to axis-angle representation The quaternion $q = (u_0, u)$ corresponds to to a counterclockwise rotation of angle θ about a unit vector ω

where θ and ω are such that

$$u_0 = \cos\frac{\theta}{2}, \text{ and } u = \sin\frac{\theta}{2}\omega. \tag{4.15}$$

So given an axis-angle representation of rotation like in Rodrigues' formula (θ, ω) we can write the quaternion as

$$q = \left(\cos\frac{\theta}{2}, \sin\frac{\theta}{2}\omega\right).$$

Using this, we can also compute the inverse of a quaternion (rotation of angle θ about the opposite axis $-\omega$) as

$$q^{-1} := \left(\cos\frac{\theta}{2}, -\sin\frac{\theta}{2}\omega\right).$$

The inverse quaternion is therefore the quaternion where all entries except the first have their signs flipped.

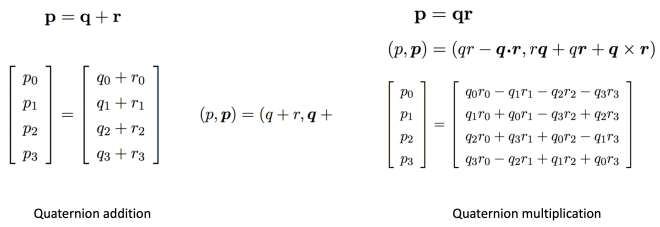


Figure 4.3: Adding two quaternions is (intuitively) defined as the element-wise sum; Multiplication, however, is defined differently from what one might suspect

Multiplication of quaternions Just like two rotation matrices multiply together to give a new rotation (rotations are a group), since quaternions are also a representation for the group of rotations, we can also multiply two quaternions $q_1 = (u_0, u), q_2 = (v_0, v)$ together using the quaternion identities for i, j, k in (4.14) to get a new quaternion

$$q_1 q_2 \equiv (u_0, u) \cdot (v_0, v) = (u_0 v_0 - u^\top v, u_0 v + v_0 u + u \times v).$$

ℹ Quaternions belong to a larger group than rotations called the Symplectic Group $Sp(1)$.

Pure quaternions A pure quaternion is a quaternion with a zero scalar value $u_0 = 0$. This is very useful to simply store a standard 3D vector $u \in \mathbb{R}^3$ as a quaternion $(0, u)$. We can then rotate points easily between different frames as follows. Given a vector $x \in \mathbb{R}^3$ we can form a quaternion $(0, x)$ and show that

$$q \cdot (0, x) \cdot q^* = (0, R(q)x). \tag{4.16}$$

where $q^* = (u_0, -u)$ is the conjugate quaternion of $q = (u_0, u)$; the conjugate is the same as the inverse for unit quaternions. Notice how the right-hand side is the vector $R(q)x$ corresponding to the vector x rotation by a matrix $R(q)$.

Quaternions to rotation matrix The rotation matrix corresponding to a quaternion is

$$\begin{aligned} R(q) &= (u_0^2 - u^\top u) I_{3\times 3} + 2\frac{u_0 u}{\|u\|} + 2uu^\top \\ &= \begin{bmatrix} 2(u_0^2+u_1^2)-1 & 2(u_1u_2-u_0u_3) & 2(u_1u_3+u_0u_2) \\ 2(u_1u_2+u_0u_3) & 2(u_0^2+u_2^2)-1 & 2(u_2u_3-u_0u_1) \\ 2(u_1u_3-u_0u_2) & 2(u_2u_3-u_0u_1) & 2(u_0^2+u_3^2)-1 \end{bmatrix}. \end{aligned} \tag{4.17}$$

Using this you can show the identity that rotation matrix corresponding to the product of two quaternions is the product of the individual rotation matrices

$$R(q_1 q_2) = R(q_1)R(q_2).$$

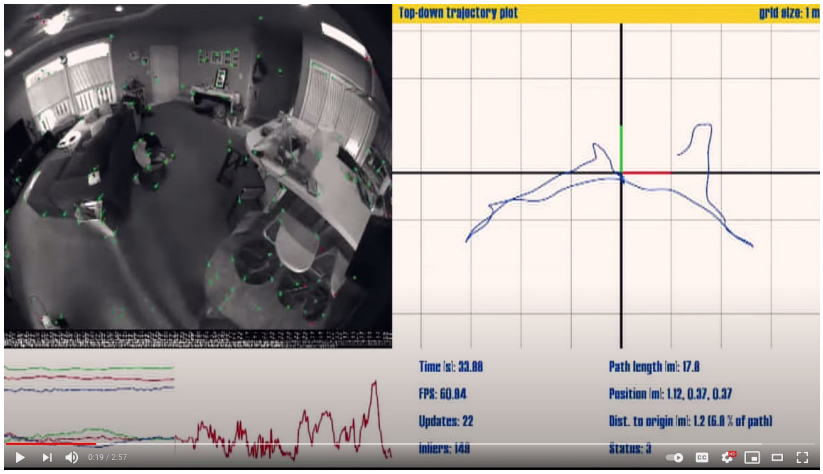
Rotation matrix to quaternion We can also go in the reverse direction. Given a rotation matrix R, the quaternion is

$$\begin{aligned} u_0 &= \frac{1}{2}\sqrt{r_{11}+r_{22}+r_{33}+1} \\ \text{if } u_0 \neq 0,\ u_1 &= \frac{r_{32}-r_{23}}{4u_0} \\ u_2 &= \frac{r_{13}-r_{31}}{4u_0} \\ u_3 &= \frac{r_{21}-r_{12}}{4u_0} \\ \text{if } u_0 = 0, u_1 &= \frac{r_{13}r_{12}}{\sqrt{r_{12}^2 r_{13}^2 + r_{12}^2 r_{23}^2 + r_{13}^2 r_{23}^3}} \\ u_2 &= \frac{r_{12}r_{23}}{\sqrt{r_{12}^2 r_{13}^2 + r_{12}^2 r_{23}^2 + r_{13}^2 r_{23}^3}} \\ u_3 &= \frac{r_{13}r_{23}}{\sqrt{r_{12}^2 r_{13}^2 + r_{12}^2 r_{23}^2 + r_{13}^2 r_{23}^3}}. \end{aligned} \tag{4.18}$$

4.3 Occupancy Grids

Rotation matrices and quaternions let us capture the dynamics of a rigid robot body. We will next look at how to better understand observations.

What is location and what is mapping? Imagine a robot that is moving around in a house. A natural representation of the state of this robot is the 3D location of all the interesting objects in the room, e.g., https://www.youtube.com/watch?v=Qe10ExwzCqk. At each time-instant, we record an observation from our sensor (in this case, a camera) that indicates how far an object is from the robot. This helps us discover the location of the objects in the room. After gathering enough observations, we would have created a *map* of the entire house. This map is the set of positions of all interesting objects in the room. Such a map is called a "feature map", these are all the green points in the image below



The main point to understand about feature map is that we can hand over this map to another robot that comes to the same house. The robot compares images from its camera and if it finds one of the objects inside the map, it can get an estimate of its location/orientation in the room with respect to the known location of the object in the map. The map is just a set of "features" that help identify salient objects in the room (objects which can be easily detected in images and relatively uniquely determine the location inside the room). The second robot using this map to estimate its position/orientation in the room is called the *localization problem*. We already know how to solve the localization problem using filtering.

The first robot was solving a harder problem called *Simultaneous Localization And Mapping (SLAM)*: namely that of discovering the location of both itself and the objects in the house. This is a very important and challenging problem in robots but we will not discuss it further. MEAM 620 digs deeper into it.

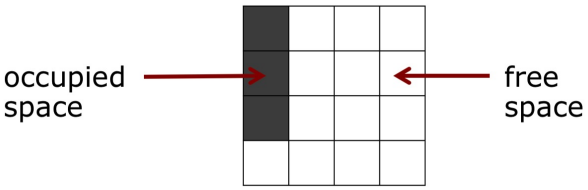
> In this section, we will solve a part of the SLAM problem, namely the mapping problem. We will assume that we know the position/orientation of the robot in the 3D world, and want to build a map of the objects in the world. We will discuss grid maps, which are a more crude way of representing maps than feature maps but can be used easily even if there are lots of objects.

Grid maps We will first discuss two-dimensional grid maps, they look as follows.

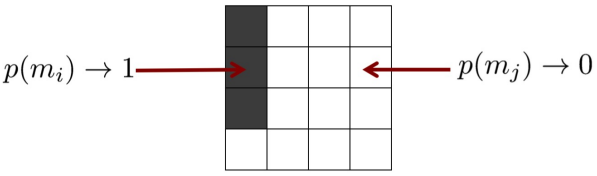
Figure 4.4: A grid map (also called an occupancy grid) is a large gray-scale image, each pixel represents a cell in the physical world. In this picture, cells that are occupied are colored black and empty cells represent free space. A grid map is a useful representation for a robot to localize in this house using observations from its sensors and comparing those to the map.

To get a quick idea of what we want to do, you can watch the mapping being performed in https://www.youtube.com/watch?v=JJhEkIA1xSE. We are interested in learning such maps from the observations that a robot collects as it moves around the physical space. Let us make two simplifying assumptions.

Assumption 1: each cell is either free or occupied



This is neat: we can now model each cell as a binary random variable that indicates occupancy. Let the probability that the cell m_i be occupied be $p(m_i)$

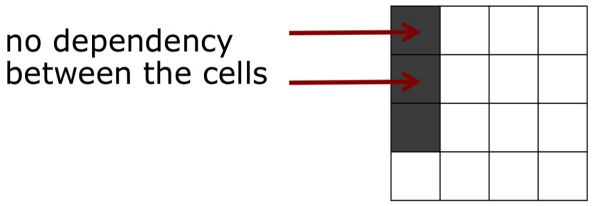


If we have $p(m_i) = 0$, then the cell is not occupied and if we have $p(m_i) = 1$, then the cell is occupied. A priori, we do not know the state of the cell so we will set the prior probability to be $p(m_i) = 0.5$.

Assumption 2: the world is static Objects in the world do not move. This is reasonable if we are interested in estimating in building a map of the walls inside the room. Note that it is not a reasonable assumption if there are moving

people inside the room. We will see a clever hack where the Bayes rule helps automatically disregard such moving objects in this section.

Assumption 3: cells are independent of each other This is another drastic simplification. The state of our system is the occupancy of each cell in the grid map. We assume that before receiving any observations, the occupancy of each individual cell is independent; it is a Bernoulli variable with probability 1/2 since we have assumed the prior to be uniform in Assumption 1.



This means that if cells in the map are denoted by a vector $m = (m_1, \dots,)$, then the probability of the cells being occupied/not-occupied can be written as

$$p(m) = \prod_i p(m_i). \tag{4.19}$$

4.3.1 Estimating the map from the data

Say that the robot pose (position and orientation) is given by the sequence $x_1, \dots, x_k$. While proceeding along this sequence, the robot receives observations $y_1, \dots, y_k$. Our goal is to estimate the state of each cell $m_i \in \{0, 1\}$ (aka "the map" $m = (m_1, m_2, \dots,)$)

$$\mathrm{P}(m \mid x_1, \dots, x_k, y_1 \dots, y_k) = \prod_i \mathrm{P}(m_i \mid x_1, \dots, x_k, y_1 \dots, y_k). \tag{4.20}$$

This is called the "static state" Bayes filter and is conceptually exactly the same as the recursive application of Bayes rule in Chapter 2 for detecting whether the door was open or closed.

We will use a short form to keep the notation clear

$$y_{1:k} = (y_1, y_2, \dots, y_k);$$

the quantity $x_{1:k}$ is defined similarly. As usual we will use a recursive Bayes

filter to compute this probability as follows.

$$\begin{aligned}
\mathrm{P}(m_i \mid x_{1:k}, y_{1:k}) &\overset{\text{Bayes rule}}{=} \frac{\mathrm{P}(y_k \mid m_i, y_{1:k-1}, x_{1:k})\,\mathrm{P}(m_i \mid y_{1:k-1}, x_{1:k})}{\mathrm{P}(y_k \mid y_{1:k-1}, x_{1:k})} \\
&\overset{\text{Markov}}{=} \frac{\mathrm{P}(y_k \mid m_i, x_k)\,\mathrm{P}(m_i \mid y_{1:k-1}, x_{1:k-1})}{\mathrm{P}(y_k \mid y_{1:k-1}, x_{1:k})} \\
&\overset{\text{Bayes rule}}{=} \frac{\mathrm{P}(m_i \mid y_k, x_k)\,\mathrm{P}(y_k \mid x_k)\,\mathrm{P}(m_i \mid y_{1:k-1}, x_{1:k-1})}{\mathrm{P}(m_i \mid x_k)\,\mathrm{P}(y_k \mid y_{1:k-1}, x_{1:k})} \\
&\overset{\text{Markov}}{=} \frac{\mathrm{P}(m_i \mid y_k, x_k)\,\mathrm{P}(y_k \mid x_k)\,\mathrm{P}(m_i \mid y_{1:k-1}, x_{1:k-1})}{\mathrm{P}(m_i)\,\mathrm{P}(y_k \mid y_{1:k-1}, x_{1:k})}.
\end{aligned} \tag{4.21}$$

We have a similar expression for the opposite probability

$$\mathrm{P}(\neg m_i \mid x_{1:k}, y_{1:k}) = \frac{\mathrm{P}(\neg m_i \mid y_k, x_k)\,\mathrm{P}(y_k \mid x_k)\,\mathrm{P}(\neg m_i \mid y_{1:k-1}, x_{1:k-1})}{\mathrm{P}(\neg m_i)\,\mathrm{P}(y_k \mid y_{1:k-1}, x_{1:k})}.$$

Let us take the ratio of the two to get

$$\begin{aligned}
\frac{\mathrm{P}(m_i \mid x_{1:k}, y_{1:k})}{\mathrm{P}(\neg m_i \mid x_{1:k}, y_{1:k})} &= \frac{\mathrm{P}(m_i \mid y_k, x_k)}{\mathrm{P}(\neg m_i \mid y_k, x_k)} \frac{\mathrm{P}(m_i \mid y_{1:k-1}, x_{1:k-1})}{\mathrm{P}(\neg m_i \mid y_{1:k-1}, x_{1:k-1})} \frac{\mathrm{P}(\neg m_i)}{\mathrm{P}(m_i)} \\
&= \underbrace{\frac{\mathrm{P}(m_i \mid y_k, x_k)}{1 - \mathrm{P}(m_i \mid y_k, x_k)}}_{\text{uses observation } y_k} \underbrace{\frac{\mathrm{P}(m_i \mid y_{1:k-1}, x_{1:k-1})}{1 - \mathrm{P}(m_i \mid y_{1:k-1}, x_{1:k-1})}}_{\text{recursive term}} \underbrace{\frac{1 - \mathrm{P}(m_i)}{\mathrm{P}(m_i)}}_{\text{prior}}.
\end{aligned} \tag{4.22}$$

This is called the odds ratio. Notice that the first term uses the latest observation y_k, the second term can be updated recursively because it is a similar expression as the left-hand side and the third term is a prior probability of the cell being occupied/non-occupied. Let us rewrite this formula using the log-odds-ratio that makes implementing it particularly easy. The log-odds-ratio of the probability $p(x)$ of a binary variable x is defined as

$$l(x) = \log \frac{p(x)}{1 - p(x)}, \text{ and } p(x) = 1 - \frac{1}{1 + e^{l(x)}}.$$

The product in (4.22) now turns into a sum as

$$l(m_i \mid y_{1:k}, x_{1:k}) = l(m_i \mid y_k, x_k) + l(m_i \mid y_{1:k-1}, x_{1:k-1}) - l(m_i). \tag{4.23}$$

This expression is used to update the occupancy of each cell. The term

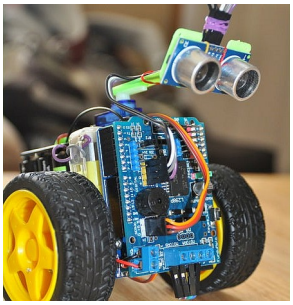
$$\text{sensor model} = l(m_i \mid y_k, x_k)$$

is different for different sensors and we will investigate it next.

❓ We assumed that the map was static. Can you think of why (4.23) automatically lets us handle some moving objects? Think of what the prior odds $l(m_i)$ does to the log-odds-ratio $l(m_i \mid y_{1:k}, x_{1:k})$.

4.3.2 Sensor models

Sonar This works by sending out an ultrasonic chirp and measuring the time between emission and reception of the signal. The time gives an estimate of the distance of an object to the robot.



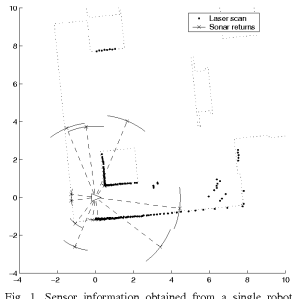
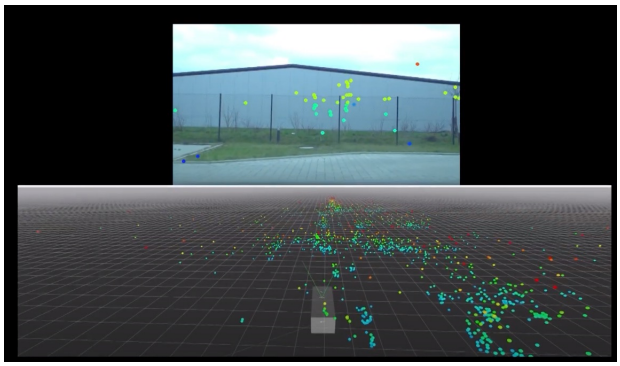


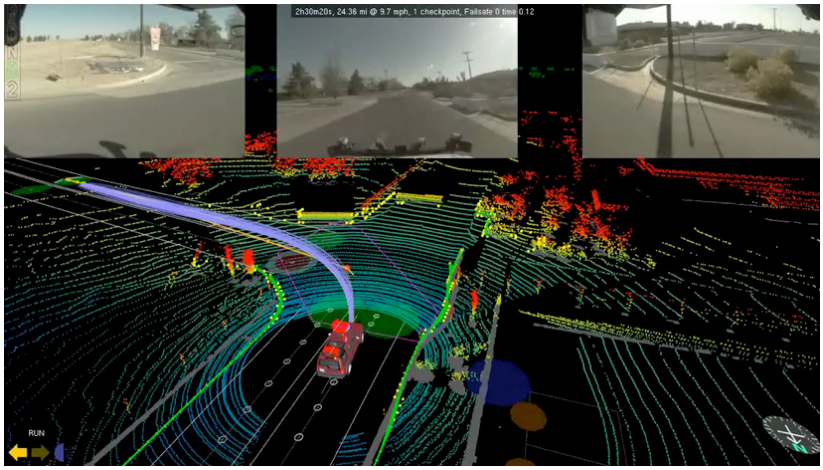
Fig. 1 Sensor information obtained from a single robot

The figure above shows a typical sonar sensor (the two "eyes") on a low-cost robot. Data from the sensor is shown on the right, a sonar is a very low resolution sensor and has a wide field of view, say 15 degrees, i.e., it cannot differentiate between objects that are within 15 degrees of each other and registers them as the same point. Sophisticated sonar technology is used today in marine environments (submarines, fish finders, detecting mines etc.).

Radar works in much the same way as a sonar except that it uses pulses of radio waves and measures the phase difference between the transmitted and the received signal. This is a very versatile sensor (it was invented by the US army to track planes and missiles during World War II) but is typically noisy and requires sophisticated processing to be used for mainstream robotics. Autonomous cars, collision warning systems on human-driven cars, weather sensing, and certainly the military use the radar today. The following picture and the video https://www.youtube.com/watch?v=hwKUcu_7F9E will give you an appreciation of the kind of data that a radar records. Radar is a very long range sensor (typically 150 m) and works primarily to detect metallic objects.



LiDAR LiDAR, which is short for Light Detection and Ranging, (https://en.wikipedia.org/wiki/Lidar) is a portmanteau of light and radar. It is a sensor that uses a pulsed laser as the source of illumination and records the time it takes (nanoseconds typically) for the signal to return to the source. See https://www.youtube.com/watch?v=NZKvf1cXe8s for how the data from a typical LiDAR (Velodyne) looks like. While a Velodyne contains an intricate system of rotating mirrors and circuitry to measure time elapsed, there are new solid state LiDARs that are rapidly evolving to match the needs of the autonomous driving industry. Most LiDARs have a usable range of about 100 m.



A typical autonomous car This is a picture of MIT's entry named Talos to the DARPA Urban Challenge (https://en.wikipedia.org/wiki/DARPA_Grand_Challenge_(2007)) which was a competition where teams had to traverse a 60 mile urban route within 6 hours, while obeying traffic laws, understanding incoming vehicles etc. Successful demonstrations by multiple teams led by (CMU, Stanford, Odin College, MIT, Penn and Cornell) in this competition jump-started the wave of autonomous driving. While the number of sensors necessary to drive well has come down (Tesla famously does not like to use LiDARs and rely exclusively on cameras and radars), the type of sensors and challenges associated with them remain essentially the same.

Waymo's autonomous car

4.3.3 Back to sensor modeling

Let us go back to understanding our sensor model $l(m_i \mid y_k, x_k)$ where m_i is a particular cell of the occupancy grid, y_k and x_k are the observations and robot position/orientation at time k.

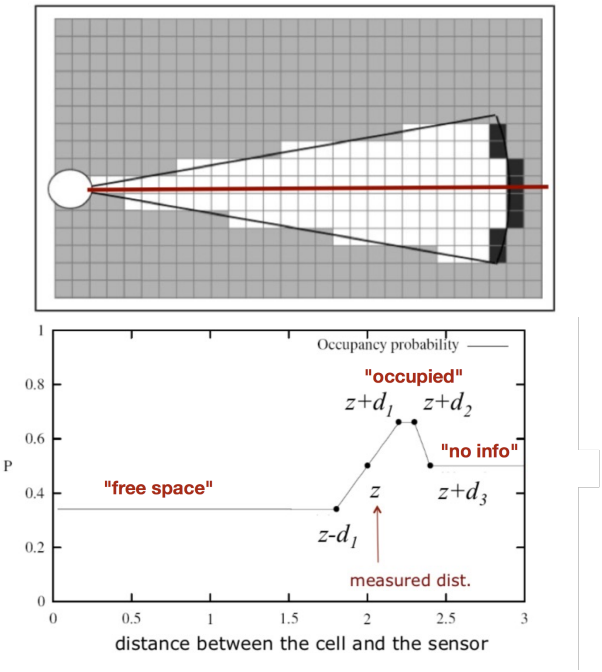
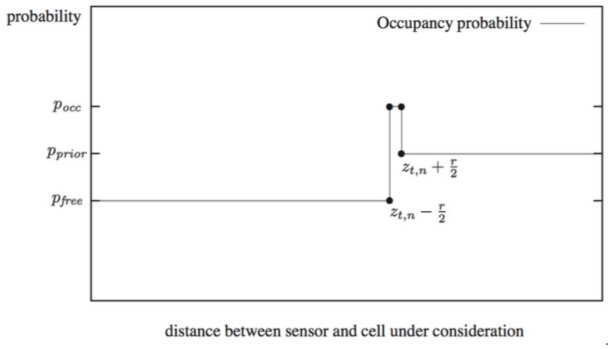


Figure 4.5: **Model for sonar data.** (Top) A sonar gives one real-valued reading corresponding to the distance measured along the red axis. (Bottom) if we travel along the optical axis, the occupancy probability $\mathrm{P}(m_i \mid y_k = z, x_k)$ can be modeled as a spike around the measured value z. It is very important to remember that range sensors such as sonar gives us three kinds of information about this ray: (i) all parts of the environment up to $\approx z$ are *unoccupied* (otherwise we would not have recorded z), (ii) there is some object at z which resulted in the return, (iii) but we do not know anything about what is behind z. So incorporating a measurement y_k from a sonar/radar/lidar involves not just updating the cell which corresponds to the return, but also updating the occupancy probabilities of every grid call along the axis.

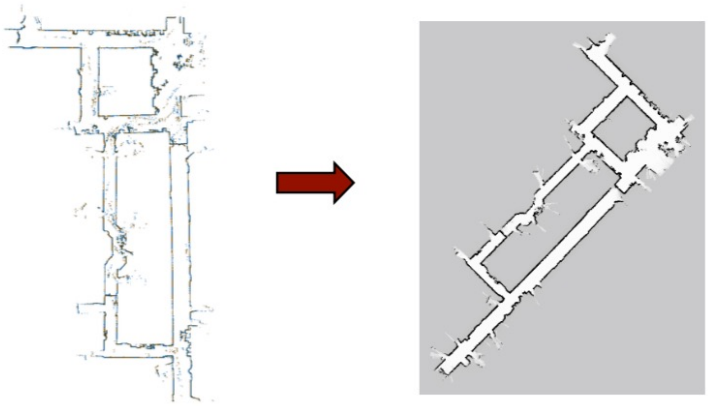
Figure 4.6: (Left) A typical occupancy grid created using a sonar sensor by updating the log-odds-ratio $l(m_i \mid x_{1:k}, y_{1:k})$ for all cells i for multiple time-steps k. At the end of the map building process, if $l(m_i \mid x_{1:k}, y_{1:k}) > 0$ for a particular cell, we set its occupancy to 1 and to zero otherwise, to get the maximum-likelihood estimate of the occupancy grid on the right.

LiDAR model When we say that a LiDAR is a more accurate sensor than the sonar, what we really mean is that the sensor model $\mathrm{P}(m_i \mid y_k, x_k)$ looks

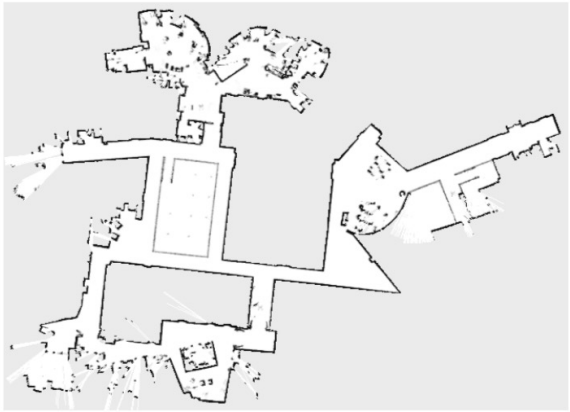
as follows.



As a result, we can create high-resolution occupancy grids using a LiDAR.



Example: MIT CSAIL 3rd Floor



❓ How will you solve the localization problem given the map? In other words, if we know the occupancy grid of a building as estimated in a prior run, and we now want to find the position/orientation of the robot traveling in this building, how show we use these sensors?

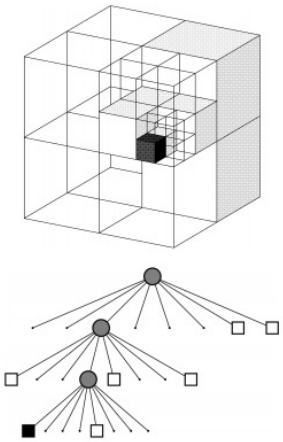
4.4 3D occupancy grids

Two-dimensional occupancy grids are a fine representation for toy problems but they run into some obvious problems: since the occupancy grid is a "top

view" of the world, we cannot represent non-trivial objects in it correctly (a large tree with a thin trunk eats up all the free space). We often desire a fundamentally three-dimensional representation of the physical world.

We could simply create cells in 3D space and our method for occupancy grid would work but this is no longer computationally cheap. For instance, if we want to build a map of Levine Hall (say 50 m $\times$ 50 m area and height of 25 m), a 3D grid map with a resolution of 5 cm $\times$ 5 cm $\times$ cm would have about 500 million cells (if we store a float in each cell this map will require about 2 GB memory). It would be cumbersome to carry around so many cells and update their probabilities after each sensor reading (a Velodyne gives data at about 30 Hz). More importantly, observe that most of the volume inside Levine is free space (inside of offices, inner courtyard etc.) so we do not really need fine resolution in those regions.

Octrees We would ideally have an occupancy grid whose resolution adapts with the kind of objects that are detected by the sensors. If nearby cells are empty we want to collapse them together to save on memory and computation, on the other hand, if nearby cells are all occupied, we want to *refine* the resolution in that area so has to more accurately discern the shape of the underlying objects. Octrees are an efficient representation for 3D volumes.

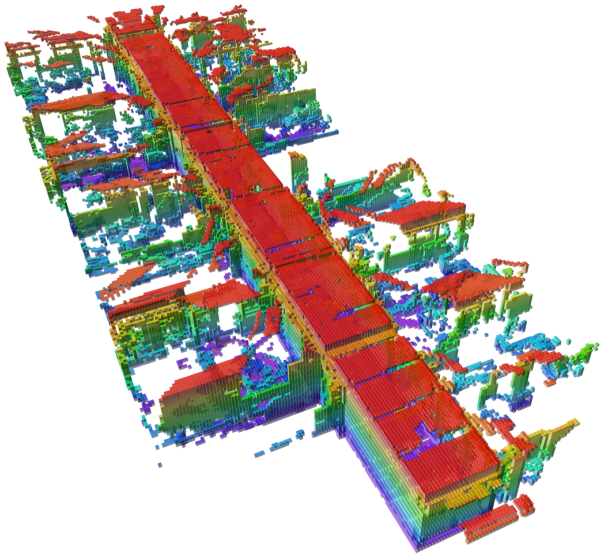


An octree is a hierarchical data structure that recursively sub-divides the 3D space into octants and allocates volumes as needed for a particular data point observed by a range sensor. It is analogous to a kd-tree. Imagine if the entire

space in the above picture were empty (the tree only has a root node), and we receive a reading corresponding to the dark shaded region. An octree would sub-divide the space starting from the root (each node in the tree populates is the parent of its eight child octants) recursively until some pre-determined minimum resolution is reached. This leaf node is grid cell; notice how different cells in the octree have different resolutions. Occupancy probabilities of each leaf node are updated using the same formula as that of (4.23). A key point here is that octrees are designed for accurate sensors (LiDARs) where there is not much noise in the observations returned by the sensor (and thereby we do not refine unnecessary parts of the space)

Octrees are very efficient at storing large map, I expect you can store the entire campus of Penn in about a gigabyte. Ray tracing (following all the cells m_i in tree along the axis of the sensor in Figure 4.5) is harder in this case but there are efficient algorithms devised for this purpose. An example OctoMap (an occupancy map created using an Octree) of a building on the campus of the University of Freiburg is shown below.

ℹ You can find LiDAR maps of the entire United States (taken from a plane) at https://www.usgs.gov/core-science-systems/ngp/3dep



4.5 Local Map

In this chapter, we primarily discussed occupancy grids of static environments as the robot moves around in the environment. The purpose of doing so is localization, namely, finding the pose of the robot by comparing the observations of the sensors with the map (think of the particle filter localization example in Chapter 3). In typical problems, we often maintain two kinds of maps, (i) a large occupancy grid for localization (say as big as city), and (ii) another smaller map, called the local map, that is used to maintain the locations of objects (typically objects that can move) in the vicinity of the robot, say a 100 m $\times$ 100 m area.

The local map is used for planning and control purposes, e.g., to check if the planned trajectory of the robot does not collide with any known obstacles. See an example of the local map at the 1:42 min mark at https://www.youtube.com/watch?v=2va15BE-7lQ. Some people also call the local map a "cost map" because occupied cells in the local map indicate a high collision cost of moving through that cell. The local map is typically constructed in the body frame and evolves as the robot moves around (objects appear in the front of the robot and are spawned in the local map and disappear from the map at the back as the robot moves forward).

> You should think of the map (and especially the local map) as the filtering estimate of the locations of various objects in the vicinity of the robot computed on the basis of multiple observations received from the robot's sensors.

Figure 4.7: The output of perception modules for a typical autonomous vehicle (taken from https://www.youtube.com/watch?v=tiwVMrTLUWg. The global occupancy grid is shown in gray (see the sides of the road). The local map is not shown in this picture but you can imagine that it has occupied voxels at all places where there are vehicles (purple boxes) and other stationary objects such as traffic light, nearby buildings etc. Typically, if we know that so and so voxel corresponds to a vehicle, we run an Extended Kalman Filter for that particular vehicle to estimate the voxels in the local map that it is likely to be in, in the next time-instant. The local map is a highly dynamic data structure that is rich in information necessary for planning trajectories of the robot.

4.6 Discussion

Occupancy grids are a very popular approach to represent the environment given the poses of the robot as it travels in this environment. We can also use occupancy grids to localize the robot in a future run (which is usually the purpose of creating them). Each cell in an occupancy grid stores the posterior probability of the cell being occupied on the basis of multiple observations $\{y_1, \ldots, y_k\}$ from respective poses $\{x_1, \ldots, x_k\}$. This

is a very efficient representation of the 3D world around us with the one caveat that each cell is updated independently of the others. But since one gets a large amount of data from typical range senors (a 64 beam Velodyne (https://velodynelidar.com/products/hdl-64e) returns about a 2 million points/sec and cheaper versions of this sensor will cost about $100), this caveat does not hurt us much in practice. You can watch this talk (https://www.youtube.com/watch?v=V8JMwE_L5s0) by the head of Uber's autonomous driving group to get more perspective about localization and mapping.