2014

A Mathematical Framework for Unmanned Aerial Vehicle Obstacle Avoidance

Sorathan Chaturapruek

*Harvey Mudd College*

---

**Recommended Citation**


https://scholarship.claremont.edu/hmc_theses/60

---

This Open Access Senior Thesis is brought to you for free and open access by the HMC Student Scholarship at Scholarship @ Claremont. It has been accepted for inclusion in HMC Senior Theses by an authorized administrator of Scholarship @ Claremont. For more information, please contact scholarship@cuc.claremont.edu.
A Mathematical Framework for Unmanned Aerial Vehicle Obstacle Avoidance

Sorathan (Tum) Chaturapruek

Weiqing Gu, Advisor

Zachary Dodds, Reader

Department of Mathematics

May, 2014
Abstract

The obstacle avoidance navigation problem for Unmanned Aerial Vehicles (UAVs) is a very challenging problem. It lies at the intersection of many fields such as probability, differential geometry, optimal control, and robotics. We build a mathematical framework to solve this problem for quadrotors using both a theoretical approach through a Hamiltonian system and a machine learning approach that learns from human sub-experts’ multiple demonstrations in obstacle avoidance. Prior research on the machine learning approach uses an algorithm that does not incorporate geometry. We have developed tools to solve and test the obstacle avoidance problem through mathematics.
## Contents

[Abstract](#) .................................................. iii

[Acknowledgments](#) ........................................ xi

1 Introduction ........................................... 1

1.1 Background ........................................ 1

1.2 Related Work ......................................... 1

1.3 Thesis Overview ..................................... 3

1.4 Key Results .......................................... 3

1.5 Structure of Thesis Report ......................... 4

2 Configuration of 3D Robot and Quadrotor Model 5

2.1 Static Model ......................................... 7

2.2 Dynamic Model ..................................... 10

2.3 SE(3) and Representation of a Three-Dimensional Moving Scene .................................................. 13

2.4 Rigid-Body Motion .................................. 14

2.5 Coordinate and velocity transformations .......... 23

2.6 Summary ............................................. 25

3 Obstacle Avoidance ...................................... 27

3.1 Related Work ......................................... 27

3.2 Environment Map Representation .................. 27

3.3 Obstacle Representation ............................. 28

3.4 Problem Setup ....................................... 29

3.5 Collision Free Path ................................ 30

3.6 Trajectory Search Algorithms ...................... 33
Bibliography 91
# List of Figures

1.1 A map constructed by an autonomous vehicle with a Kinect, from the MIT, University of Washington, and Intel research labs (click link) .................................................. 2

2.1 A torus, as an example of a differential manifold ..................... 6

2.2 Base image is a CyberFlight Dominator, taken from DirectIndu-

try (2013) .......................................................................................... 10

2.3 Differential Flatness. Image from Pratik Chaudhari, “Ag-
gressive maneuvers using differential flatness for a quadro-
tor” .................................................................................................. 11

2.4 A rigid transformation, which preserves the distance between
any two points .................................................................................. 15

2.5 A rigid-body motion between a camera frame \( C : (x, y, z) \)
and a world coordinate frame \( W : (X, Y, Z) \) .................................. 15

2.6 Rotation of a rigid body about a fixed point \( o \) and along the
axis \( \omega \). The coordinate frame \( W \) (solid line) is fixed, and
the coordinate frame (dashed line) is attached to the rotat-
ing rigid body ................................................................................. 17

2.7 Composition of rigid-body motions ........................................... 24

3.1 A cylinder obstacle problem setup. The start point is at \((-a, 0, h)\).
The end point is at \((b, 0, h)\), where \(a, b > 1\) and \(h > 0\). The
cylinder is \( \{(x, y, z) \in \mathbb{R}^3 | x^2 + y^2 = 1, 0 \leq z \leq h]\} \) .... 29

3.2 An example of the Minkowski sum ............................................. 30

3.3 An optimal length collision-free path ....................................... 32

3.4 A drone obstacle avoidance problem ......................................... 32

3.5 Hybrid automata in action. The wiggle lines represent a drift. 34

3.6 Hybrid automata model schema. GTG is “Go to Goal”. AO is
“Avoid Obstacles” ................................................................. 34
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.7</td>
<td>Image from Pratik Chaudhari, &quot;Aggressive maneuvers using differential flatness for a quadrotor&quot;</td>
<td>35</td>
</tr>
<tr>
<td>6.1</td>
<td>An example of polynomial curve fitting onto data. The black curve is the best polynomial curve of degree 7 that fits the data.</td>
<td>57</td>
</tr>
<tr>
<td>7.1</td>
<td>AR.Drone 2.0 from various points of view, and with and without the indoor protective hull.</td>
<td>62</td>
</tr>
<tr>
<td>7.2</td>
<td>A Logitech joystick for an easier control of the drone.</td>
<td>66</td>
</tr>
<tr>
<td>7.3</td>
<td>Parallel Tracking and Mapping (PTAM) and state estimation. The top, middle, and bottom pictures show the beginning, middle, and ending of the feature point tracking process.</td>
<td>68</td>
</tr>
<tr>
<td>7.4</td>
<td>Point cloud after feature point tracking.</td>
<td>69</td>
</tr>
<tr>
<td>7.5</td>
<td>Horizontal plane drone navigation problem.</td>
<td>70</td>
</tr>
<tr>
<td>8.1</td>
<td>A screenshot of a quadrotor model in X-plane. The quadrotor model QRO_X in X-plane is provided by JLN.</td>
<td>75</td>
</tr>
<tr>
<td>A.1</td>
<td>(top) A Kinect; (bottom) A screenshot of the output of Kinect of Tum waving his hand. The gradient of color shows the depth of objects in the image.</td>
<td>90</td>
</tr>
</tbody>
</table>
Acknowledgments

I am very grateful to Weiqing Gu for her invaluable support, patience, friendship, insightful guidance and mentorship that extend far beyond what a thesis document can describe. My senior year at Mudd would not be as memorable as it is without her. Her enthusiasm is unmatched and contagious. I would like to thank Zachary Dodds for practical advice in experimenting with quadrotors. Thank Andrew P. Turner for working on Optimal Control on SE(3) together. Thank you all mathematics professors who taught me to become a better mathematician and other people who make my experience at Mudd so special. In particular, I would like to thank Michael Orrison to help me learn how to think mathematically and make every part of mathematics meaningful; Andrew Bernoff for his endless support, his insightful Putnam guidance and his contributions that make my first two summers possible (at PCMI and UCLA, respectively); Dagan Karp for helping me appreciate abstract and elegant concepts, for inspiring my curious minds, and for really supporting me to do well at Mudd; Francis Su for letting me in the Cauchy Critters society, “an elite group with a really silly name”, for his thoughtful advice throughout my career and Mudd, and for his examples of giving a very engaging talk in every single talk of his that I attended; Wendy Menefee-Libey for giving me the two best jobs on campus: writing center consultant and academic excellence facilitator in math; Melissa O’Neill for making me realize that I do not have to be perfect at everything; Peter Saeta for teaching me how to write, being my awesome host family, and working together in a Clinic project to make Mudd physics a massive open online course; Thank you my family and my friends for their incredible support throughout my sometimes sleepless nights and distinctly memorable time at Mudd.
Chapter 1

Introduction

1.1 Background

Unmanned aerial vehicles (UAVs) are robots that infer their position and orientation from data collected by constantly changing the UAV’s rotational orientation. Quadrotor helicopters are UAVs that have four rotors aligned in a rectangle. They are popular among the UAV research community because of the simplicity of their construction and maintenance, their ability to hover, and their vertical take off and landing (VTOL) capability (see Hoffmann et al. (2007)).

Because of their powerful and specific capabilities, many applications have been envisaged both as individual vehicles and in multiple vehicle teams. For example, we might use the UAV for aiding a disabled person at home, for search and rescue, for surveillance, and even for entertainment.

1.2 Related Work

There have been many active UAV research activities from various universities and research institutes who approach the problem from very different aspects, ranging from the control design to the implementation and testing.

At the University of Pennsylvania, led by Professor Vijay Kumar, the group has developed a swarming model and navigation technology. A demonstration of a developed model of multi-floor indoor navigation can be viewed here (see Shen et al. (2011)).

\footnote{This Youtube video shows KMel quadrotors dance to music and lights at Saatchi and Saatchi event in Cannes.}

\footnote{http://www.youtube.com/watch?v=TjQPHprBTps}
After the floor map has been modeled, it is natural to attempt to model indoor obstacles. The Robust Robotics Group of the Computer Science and Artificial Intelligence Laboratory at MIT, the Robotics and State Estimation Laboratory at University of Washington, and Intel Labs in Seattle collaborated to develop an autonomous flight and mapping algorithm that uses the Microsoft Kinect® to model an indoor environment. A demonstration video can be viewed below.\footnote{http://www.youtube.com/watch?v=aiNX-vpDhMo}

Here at HMC we also have a robotics lab led by Professor Dodds. His research project is Vision-based Aerial Mapping, and his REU group in 2012 built top-to-bottom systems for human-scale navigation within Mudd’s maze-like Libra Complex, which is a series of underground classrooms. The robot model is the AR.Drone 2.0 quadcopter, which we used to experimentally test of our quadrotor model. Further descriptions of the REU project can be found below.\footnote{http://www.cs.hmc.edu/reu/projects/robotics/}

As for related work that is specific to each chapter, we will discuss it at the beginning of those chapters.
1.3 Thesis Overview

The focus of this thesis project is to develop a mathematical framework for designing an intelligent autonomous 3D robot in indoor, GPS-denied environments to navigate and avoid static objects such as tables.

Autonomous flights in GPS-denied environments are desirable for many applications, such as chemical leak detection in a building. Obstacle avoidance is a fundamental and important aspect of autonomous UAV to make a successful flight and complete its missions. Given a mission to go from a start point and end point with an obstacle in-between, an intelligent robot should be able to perform the high-level goal by avoiding a collision with such obstacle. Many ground robotics problems have been well researched, but the UAV navigation and obstacle avoidance problem presents a new challenge due to its low stability, high agility, and its payload limit, and the constraint that all computational executions must be fast to prevent the UAV from falling or drifting. The limit on payload means that we cannot easily incorporate advanced sensors such as a 3D depth sensing device.

There are many fundamental questions involved in approaching obstacle avoidance problem for a UAV. How can it know its current state? How can it represent its environments? How can it recognize an obstacle? When obstacle is found, what should it do? How can it optimally avoid obstacle and move to the goal? Prior research usually framed these questions in the probability and statistics framework, but this approach does not take an advantage of the geometry and the intrinsic symmetry in the problem. We would like to approach this robot navigation and obstacle avoidance problem from a geometric point of view. In particular, we use various ideas from differential geometry such as tangent bundles, curvatures, Lie groups, to answer many of the questions above.

1.4 Key Results

We considered various robotics control problems and techniques in the context of obstacle avoidance. After the robot motion planning problem was reduced to a trajectory planning problem through the Minkowski sum, we reviewed theoretical and heuristic approaches to trajectory planning and optimal control, and then we proposed several geometric ideas to extend those approaches. More specifically, we considered how to frame the quadrotor obstacle avoidance problem in a differential geometry language, determined how to incorporate the curve curvature into the model, and
showed how to relate Lie groups to the problem. Finally, we did preliminary testing of the framework concept using AR.Drone 2.0 quadcopters.

1.5 Structure of Thesis Report

In Chapter 2 (Configuration of 3D Robot and Quadrotor Model), we first consider what information is necessary to capture UAVs’ current and future behaviors. Then we focus on the obstacle avoidance problem in Chapter 3 (Obstacle Avoidance). After we reduce the robot motion problem into a trajectory planning problem, we overview a few important pieces of the research in the UAVs optimal control area, both from a Hamiltonian approach, as in Chapter 4 (Optimal Control of 3D Robot on $SE(3)$), and a Machine learning approach, as in Chapter 5 (Learning for Control from Multiple Demonstrations). We proposed a few geometric ideas to solve the obstacle avoidance problem in Chapter 6 (Theories Meet Practices). Finally, we setup an experiment with a quadcopter model AR.Drone 2.0, where the details of set up can be found in Appendix A (Setup of Experiments with AR.Drone 2.0), and the results can be found in Chapter 7 (Results and Experiments with AR.Drone 2.0). Furthermore, we provide Appendix A (Set-up of Experiments with AR.Drone 2.0) for those who would like to set up the system to fly a drone from the ground up as a playground to run various obstacle avoidance algorithms.
Chapter 2

Configuration of 3D Robot and Quadrotor Model

In this chapter, we begin by investigating both static and dynamic models of quadrotors, and then we introduce high-level ideas for control principles.

We used a tangent bundle to describe the state of a quadrotor. Before we can talk about tangent bundles, we need to define the notion of differential manifolds, tangent vectors, and tangent spaces.

**Definition 1 (Differential Manifolds)** A differential manifold of dimension $n$ is a set $M$ and a family of injective mappings $x_\alpha : U_\alpha \subset \mathbb{R}^n \rightarrow M$ of open sets $U_\alpha$ of $\mathbb{R}^n$ into $M$ such that:

(a) $\bigcup_\alpha x_\alpha(U_\alpha) = M$.

(b) for any pair $\alpha, \beta$ with $x_\alpha(U_\alpha) \cap x_\beta(U_\beta) = W \neq \emptyset$, the sets $x_\alpha^{-1}(W)$ and $x_\beta^{-1}(W)$ are open sets in $\mathbb{R}^n$ and the mappings $x_\beta^{-1} \circ x_\alpha$ are differentiable.

Intuitively, this means that, if we zoom-in enough (locally look) at a particular point, a set of points on the manifold (neighborhood) near the point will “look like” a Euclidean space $\mathbb{R}^n$. The “look like” part is formalized by the notion of a differential injective mapping. For example, the Earth surface is a sphere. When we look at the ground around us, we think of it as $\mathbb{R}^2$. Another example of a manifold is a torus, which has a donut shape (see Figure 2.1). The condition (b) is to guarantee that all these local mappings are properly glued together.

Now that we have a manifold, we define a curve.
Definition 2 (Parametrized Curve) A parametrized differentiable curve \( \alpha(t) \) is a differentiable map \( \alpha(t) : I \rightarrow \mathbb{R}^3 \) of an open interval \( I = (a, b) \) of the real line into \( \mathbb{R}^3 \).

Recall that a map \( \alpha(t) = (x(t), y(t), z(t)) \) at \( t_0 \) is differentiable if and only if all of \( x(t), y(t), z(t) \) are differentiable at \( t_0 \). The map \( \alpha(t) \) is said to be differentiable if it is differentiable at every \( t \) in \( I \). Note that we denote \( \alpha'(t) = (x'(t), y'(t), z'(t)) \) the first derivative of \( \alpha \) at \( t \). \( \alpha'(t) \) is called the tangent vector of the curve \( \alpha \) at \( t \).

When a curve lies on a manifold, it can induce tangent vectors on a manifold. We then define the notion of tangent vectors in another way that does not rely on the ambient space, and hence allow these objects to be flexible as to which ambient spaces we embed them. For example, some objects are not embeddable in \( \mathbb{R}^3 \) and are bound to have self intersections if we place them in \( \mathbb{R}^3 \). In particular, the Klein bottle is not embeddable in \( \mathbb{R}^3 \) but is embeddable in \( \mathbb{R}^4 \). See [do Carmo (1992) (Example 4.9(b)) for more discussion on an embedding and the Klein bottle. For now we simply assume the merit in defining tangent vectors that are independent of the ambient space, as shown below.

Definition 3 (Tangent Vectors) Let \( M \) be a differentiable manifold and let \( \alpha : (-\epsilon, \epsilon) \rightarrow M \) be a differentiable curve in \( M \) with \( \alpha(0) = p \in M \), and let \( D \) be the set of functions on \( M \) that are differentiable at \( p \). The tangent vector to the curve \( \alpha \) at \( t = 0 \) is a function \( \alpha'(0) : D \rightarrow \mathbb{R} \) given by

\[
\alpha'(0)f = \frac{d}{dt}(f \circ \alpha) \bigg|_{t=0},
\]

Figure 2.1 A torus, as an example of a differential manifold.
where $f \in D$. We view $\alpha'(0)$ as a type of form, which is a function that takes as input another function and gives as output a number. A tangent vector at $p$ of $M$ is then a tangent vector at $t = 0$ of some curve $\beta : (-\epsilon, \epsilon) \to M$ where $\beta(0) = p$.

Now that we have tangent vectors on a manifold, we are ready to define a tangent space, which is simply a collection of tangent vectors.

**Definition 4 (Tangent Spaces)** Given a manifold $M$ and a point $p \in M$, the tangent space $T_p(M)$ is the set of all tangent vectors at the point $p$ of the differentiable manifold $M$.

We can attach a tangent space to each of the points on a manifold. This notion is captured in the tangent bundle definition:

**Definition 5 (Tangent Bundles)** The tangent bundle of a differential manifold $M$ is

$$TM = \bigcup_{x \in M} \{x\} \times T_xM = \bigcup_{x \in M} \{(x, y) | y \in T_xM\},$$

where $T_xM$ denotes the tangent space to $M$ at point $x$.

And now we will apply these notions in the context of robotics.

### 2.1 Static Model

First, we define the state of a quadrotor to be

$$x = [x, y, z, \phi, \theta, \psi, \dot{x}, \dot{y}, \dot{z}, p, q, r]^T,$$

where $\phi, \theta, \psi$ are roll, pitch, yaw, respectively, and $p, q, r$ are coefficients that represent the angular velocity of the body frame. We need both the position $(x, y, z)$, orientation $(\phi, \theta, \psi)$ and their corresponding derivatives in order to fully describe the current state of a quadrotor (see Mellinger and Kumar (2011)).

Let us try to connect this quadrotor state representation and ideas in differential geometry that we introduced in the first section. First, let us focus on the position and the orientation parts. Let $y = [x, y, z, \phi, \theta, \psi]$ be the first six states of $x$. We can see rephrase this as

$$y \in \text{SO}(3) \times \mathbb{R}^3,$$
8 Configuration of 3D Robot and Quadrotor Model

where

\[
\text{SO}(3) = \{ A \in M_{3 \times 3}(\mathbb{R}) | A^T A = I, \det(A) = 1 \}
\subset \text{O}(3) = \{ A \in M_{3 \times 3}(\mathbb{R}) | A^T A = I \}.
\]

is the 3-dimensional rotation group (or a special orthogonal group). Note that the column vectors correspond to coordinate axes, because the condition \( A^T A = I \) simply means that the three column vectors are orthonormal. Without the condition \( \det(A) = 1 \), as in the orthogonal group \( \text{O}(3) \), we would only have

\[
1 = \det(I) = \det(A^T A) = \det(A) \det(A^T) = \det(A)^2,
\]

implying that \( \det(A) = \pm 1 \). In fact, geometrically, \( \text{SO}(3) \) is one of the two pieces of \( \text{O}(3) \) that corresponds to positive orientation in rotation (one that preserves the sign of the determinant of three orthonormal vectors).

The \( \rtimes \) symbol is semidirect product, and \( \mathbb{R}^3 \) represents the position \( x, y, z \) (alternatively, a translation from the origin to such point). We use the semidirect product instead of the direct product because translation and rotations do not commute.

It is well known that the special orthogonal group is a manifold of dimension 3, which can be described uniquely by the roll, pitch, yaw angles. However, parametrizing the special orthogonal group by these three angles (called the Euler angles) leads to a problem called the Gimbal lock. In the next subsection, we will discuss this problem and a solution, which is to parametrize the special orthogonal group by the unit quaternions instead.

Next, in order to incorporate derivatives of positions and orientations, we have already defined the tangent bundle that precisely captures this notion. Thus, we can view the complete state as

\[
x \in T(\text{SO}(3) \rtimes \mathbb{R}^3),
\]

where \( T \) is the tangent bundle. It is also well known that a tangent bundle of a manifold is a manifold, so \( T(\text{SO}(3) \rtimes \mathbb{R}^3) \) is a manifold as well.

2.1.1 Parametrization

Parametrization by Euler Angles

The euler angles refer to roll, pitch, and yaw, which correspond to rotation in the \( Y, X, \) and \( Z \) axis in the \( X - Y - Z \) frame.
A good point about parametrization by Euler angles is that this representation is minimal, since it has 3 variables for the 3-dimensional manifold.

However, using Euler angles to parametrize $SO(3)$ can lead to **Gimbal lock**. It is a scenario in which all axes are aligned, and hence we lose the one degree of freedom of the parametrization, occurring when the axes of two of the three gimbals are driven into a parallel configuration, “locking” the system into rotation in a degenerate two-dimensional space instead. Fortunately, this problem can be remedied by using the unit quaternions to parametrize the space.

**Parametrization by Unit Quaternions**

There are four parameters $\mathbf{\hat{n}}, \theta$ and we can think of this representation as rotating by $\theta$ around the vector $\mathbf{\hat{n}}$.

$$
\mathbf{q} = (q_x, q_y, q_z, q_w)^T \in \mathbb{R}^4, \|\mathbf{q}\| = 1.
$$

We note that $\mathbf{q} = -\mathbf{q}$, so the parametrization covers $SO(3)$ twice. A good point is that multiplication and inversion operators are efficient.

**2.1.2 State Estimation**

**Input and Output Pipeline**

There are several sensors we can put on the quadrotor. Note that it has a limitation on the payload, so the sensors have to have a light weight.

- Depth-scan cameras (such as the Kinect camera)
- Accelerometer
- Gyroscope

The intermediate output is point cloud, which is a set of data points. The next step is called **registration**. This can combine with external knowledge such as color images. Then we can start processing data, including removing unnecessary info and using filters to probabilistically determine geometric features.
2.2 Dynamic Model

From the Newton’s second law of motion, we get

\[ m\ddot{r} = T_{\text{total}} R^T \cdot z - mgz, \quad (2.1) \]

where \( R \in \text{SO}(3) \) and \( r \in \mathbb{R}^3 \), and where \( T_{\text{total}} \) is the sum of the thrust produced by each rotor. Further, assuming that the body dynamics is significantly slower than the motor dynamics, we can derive

\[ I\dot{\omega}_B + \omega_B \times I\omega_B = \begin{bmatrix} u_2 \\ u_3 \\ u_4 \end{bmatrix}, \quad (2.2) \]

where \( I \) is the moment of inertia matrix referenced to the center of mass along the \( x_B-y_B-z_B \) axes.

2.2.1 Differentially Flat Systems

Fliess et al. (1995) first introduced the concept of differential flatness. For our purpose, there are two main ideas regarding the differential flatness systems. First, if a system is differentially flat, we can use the trajectory to uniquely determine the states and control. Second, the quadrotor system
is also differentially flat, which allows us to reduce a 12-dimensional state space to a 4-dimensional space, making the real-time computation more practical.

**Differential Flatness**

**Definition 6 (Differential Flatness)** A nonlinear system

\[ \dot{x} = f(x, u), \quad x \in \mathbb{R}^n, \quad u \in \mathbb{R}^m \]

is differentially flat if there exist flat outputs

\[ z = \Psi_z(x, u, u^{(1)}), \quad z \in \mathbb{R}^m \]

such that

\[ x = \Psi_x(z, \dot{z}, \ldots, z^{(l)}) \]

\[ u = \Psi_u(z, \dot{z}, \ldots, z^{(l)}) \]

where \( z^{(k)} \) denotes the \( k \)-th derivative of \( z \).
Consider the simple pendulum example for a differentially flat system. The system equations are
\[ m l^2 \ddot{\theta} + b \dot{\theta} + m g l \sin \theta = u \]
\[ z = \theta \]
\[ \theta = \Psi_{\theta}(z) = z \]
\[ u = \Psi_u(z, \dot{z}, \ddot{z}) = m l^2 \ddot{z} + b \dot{z} + m g l \sin z. \]
That is, given a trajectory \( z(t) \), we can uniquely determine the states \( \theta(t) \) and \( \dot{\theta}(t) \) and the control \( u(t) \). Thus, differential flatness gives us a way to convert trajectory planning to control and state estimation.

**Quadrotor is Differentially Flat**

The states of the system are
\[ [x, y, z, \phi, \theta, \psi, \dot{x}, \dot{y}, \dot{z}, p, q, r]^T, \]
where \( \phi, \theta, \psi \) are the Euler angles roll, pitch, yaw, respectively for the Z-X-Y frame, and \( p, q, r \) are the rates of rotation in the body frame.

Consider the following flat output
\[ \sigma = [x, y, z, \psi]^T, \]
where \( \psi \) is the yaw angle. We show that every component of \( x \) can be written as an algebraic expression of \( \sigma \) or its derivatives.

Define \( t = [\dot{x}, \dot{y}, \dot{z} + g]^T \) and \( x_C = [\cos \psi, \sin \psi, 0]^T \).

From the equation of motion (2.1), we get
\[ z_B = \frac{t}{\|t\|}, \]
\[ y_B = \frac{z_B \times x_C}{\|z_B \times x_C\|}, \]
\[ x_B = y_B \times z_B, \]
provided that \( z_B \times x_C \neq 0 \). The reason that we can use \( x_C \) to take the cross product with \( z_B \) is because we know that all of \( z_B, x_B, x_C \) live in the same plane. Thus, as long as the two vectors are not parallel, it does matter which of the two vectors we use to generate the unit vector that is normal to the plane.
In [Mellinger and Kumar (2011)], the detail of getting the Euler angles from

$$R_B = [x_B, y_B, z_B]$$

using algebraic manipulations is shown. Further, since

$$w_B = px_B + qy_B + rz_B,$$

the expressions for \( p, q, r \) can be obtained by differentiating the equation of motion (2.1). From Equation (2.1), we get

$$u_1 = m\|t\|$$

and we get \([u_2, u_3, u_4]^T\) from Equation (2.2).

### 2.3 SE(3) and Representation of a Three-Dimensional Moving Scene

#### 2.3.1 Motivation

One of the fundamental problems in robotics is how to infer a 3D map from 2D pictures taken by a moving camera. To answer this question, we need to have a way to represent a camera movement. Since a camera is not a point but an object, it seems at first that we need to keep track where each point of the camera moves. However, fortunately, because we also know that a camera is a rigid body, which means the distance between two specific points of camera is the same regardless of how the camera moves, we can keep track much less information in order to fully recover how the camera moves. We will discuss how each abstract notion translates to a more concrete representation via matrices. Many of the figures in this section and the following contents come from [Ma et al. (2005)].

#### 2.3.2 Three-Dimensional Euclidean Space

We use \( \mathbb{E}^3 \) to denote the familiar three-dimensional Euclidean space, where in general a Euclidean space is a set whose elements satisfy the five axioms. However, analytically, three-dimensional Euclidean space can be represented globally by a Cartesian frame. That is, every point \( p \in \mathbb{E}^3 \) can be identified with a point in \( \mathbb{R}^3 \) with three coordinates

$$X \overset{\text{def}}{=} [X_1, X_2, X_3]^T = \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} \in \mathbb{R}^3.$$
Then, to measure distances and angles, $E^3$ must be endowed with a metric, which can be determined by the notion of an inner product. By an appropriate choice of Cartesian frame, any inner product in $E^3$ can be converted to the following canonical form

$$\langle u, v \rangle \overset{\text{def}}{=} u^T v = u_1 v_1 + u_2 v_2 + u_3 v_3,$$

for all $u, v \in \mathbb{R}^3$.

### 2.3.3 Isomorphism between $so(3)$ and $\mathbb{R}^3$

We first consider the cross product of two vectors $u = [u_1, u_2, u_3]^T, v = [v_1, v_2, v_3]^T \in \mathbb{R}^3$, which is given by

$$u \times v \overset{\text{def}}{=} \begin{bmatrix} u_2v_3 - u_3v_2 \\ u_3v_1 - u_1v_3 \\ u_1v_2 - u_2v_1 \end{bmatrix} \in \mathbb{R}^3.$$

Notice that, if we fix $u$, the cross product induces a map from $\mathbb{R}^3$ to $\mathbb{R}^3$:

$$u \times v = \begin{bmatrix} u_2v_3 - u_3v_2 \\ u_3v_1 - u_1v_3 \\ u_1v_2 - u_2v_1 \end{bmatrix} = \begin{bmatrix} 0 & -u_3 & u_2 \\ u_3 & 0 & -u_1 \\ -u_2 & u_1 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \hat{u} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix},$$

where we have defined the hat map $\wedge : \mathbb{R}^3 \rightarrow \mathbb{R}^{3 \times 3}$ by

$$u = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \mapsto \hat{u} \overset{\text{def}}{=} \begin{bmatrix} 0 & -u_3 & u_2 \\ u_3 & 0 & -u_1 \\ -u_2 & u_1 & 0 \end{bmatrix}.$$

We can see that this is a 1-1 correspondence. The inverse map, the vee map $\vee : \mathbb{R}^{3 \times 3} \rightarrow \mathbb{R}^3$ is given by

$$\hat{u} = \begin{bmatrix} 0 & -u_3 & u_2 \\ u_3 & 0 & -u_1 \\ -u_2 & u_1 & 0 \end{bmatrix} \mapsto (\hat{u})^\vee = u = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}.$$

Therefore, we can see that $so(3)$ is isomorphic to $\mathbb{R}^3$.

### 2.4 Rigid-Body Motion

As introduced in the introduction, fortunately, because we know that a camera is a rigid body, we need not specify the motion of every point. Instead, it is sufficient to specify the motion of one point and the motion of three coordinate axes attached to that point (see Figure 2.4).
Figure 2.4  A rigid transformation, which preserves the distance between any two points.

Figure 2.5  A rigid-body motion between a camera frame $C : (x, y, z)$ and a world coordinate frame $W : (X, Y, Z)$.
2.4.1 Rotation Motion and Its Representations

Orthogonal Matrix Representation of Rotations

Suppose we have a rigid object rotating about a fix point \( o \in \mathbb{E}^3 \). How do we describe its orientation relative a chosen coordinate frame, say \( W \)? Without loss of generality, we assume that the origin of the world frame is the center of rotation. We now attach another coordinate from, say \( C \), to the rotating object, say a camera, with its origin also at \( o \) (See Figure 2.6). The orientation of the frame \( C \) relative to the frame \( W \) is determined by the coordinates of the three orthonormal vectors

\[
r_1 = g_*(e_1), r_2 = g_*(e_2), r_3 = g_*(e_3) \in \mathbb{R}^3
\]

relative to the world frame \( W \). The three vectors \( r_1, r_2, r_3 \) are also simply unit vectors along the three principal axes \( x, y, z \) of the frame \( C \), respectively. Thus, the configuration of the rotating object is then completely determined by the \( 3 \times 3 \) matrix

\[
R_{wc} \overset{\text{def}}{=} [r_1, r_2, r_3] \in \mathbb{R}^{3\times3}.
\]

with \( r_1, r_2, r_3 \) stacked in order as its three columns. Since they form an orthonormal frame, it follows that

\[
r_i^T r_j = \delta_{ij} \overset{\text{def}}{=} \begin{cases} 
1 & \text{for } i = j, \\
0 & \text{for } i \neq j.
\end{cases}
\]

which can be written compactly as

\[
R_{wc} R_{wc}^T = R_{wc} R_{wc}^T = I.
\]

Note that from this we get \( R_{wc}^{-1} = R_{wc}^T \). Because \( r_1, r_2, r_3 \) must for a right-handed frame, we further require that the determinant must be \(+1\) (the determinant in this case is an oriented volume of the parallelepiped formed by \( r_1, r_2, r_3 \)). We call a transformation special if it is orientation preserving.

Definition 7 (SO(3))

\[
\text{SO}(3) \overset{\text{def}}{=} \text{the set of the orthogonal matrices in } \mathbb{R}^{3\times3}
\]

Algebraically,

\[
\text{SO}(3) = \{ R \in \mathbb{R}^{3\times3} | R^T R = I, \det(R) = +1 \}.
\]
Figure 2.6 Rotation of a rigid body about a fixed point \( o \) and along the axis \( \omega \). The coordinate frame \( W \) (solid line) is fixed, and the coordinate frame (dashed line) is attached to the rotating rigid body.

**Canonical Exponential Coordinates for Rotations**

The \( 3 \times 3 = 9 \) parameter entries in the definition of \( SO(3) \) are not independent, due to the constraint \( R^T R = I \). We will consider an explicit parametrizations for the space of rotation matrices.

Given a trajectory \( R(t) : \mathbb{R} \rightarrow SO(3) \) that describes a continuous rotational motion, the rotation must satisfy

\[
R(t)R^T(t) = I.
\]

Computing the derivative of the above equation with respect to \( t \), we get

\[
\dot{R}(t)R^T(t) = -(\dot{R}(t)R^T(t))^T,
\]

which means that \( \dot{R}(t)R^T(t) \) is a skew-symmetric matrix. By the isomorphism lemma we proved before, there must exist a vector \( \omega(t) \in \mathbb{R}^3 \) such that

\[
\dot{R}(t)R^T(t) = \hat{\omega}(t).
\]

Multiplying both sides by \( R(t) \) on the right yields

\[
\dot{R}(t) = \hat{\omega}(t)R(t).
\]
Definition 8 (so(3))

\[
so(3) \overset{\text{def}}{=} \text{the set of the skew-symmetric matrices in } \mathbb{R}^{3 \times 3}
\]

Algebraically,

\[
so(3) = \{ \hat{\omega} \in \mathbb{R}^{3 \times 3} | \omega \in \mathbb{R}^{3} \}
\]

If \( R(0) = I \), then we have the solution

\[
R(t) = e^{\hat{\omega}t}.
\]

We can confirm that this is a rotation matrix. Thus, we have defined the exponential map \( \exp : so(3) \rightarrow SO(3) \) defined by \( \hat{\omega} \mapsto e^{\hat{\omega}} \).

A way to quickly compute the exponential of a map is by the Rodrigues’ formula.

**Proposition 1 (Rodrigues’ formula for a matrix)** Given \( \omega \in \mathbb{R}^{3} \), the matrix exponential \( R = e^{\hat{\omega}} \) is given by

\[
e^{\hat{\omega}} = I + \frac{\hat{\omega}}{\theta} \sin(\theta) + \frac{\hat{\omega}^2}{\theta^2} (1 - \cos(\theta)),
\]

where \( \theta = \| \hat{\omega} \| \).

This statement can be proved by using Taylor expansions of matrix exponentiation.

**Remark** It is useful to note that the exponential map is not commutative. That is, for two \( \hat{\omega}_1, \hat{\omega}_2 \in so(3) \), \( e^{\hat{\omega}_1} e^{\hat{\omega}_2} \neq e^{\hat{\omega}_2} e^{\hat{\omega}_1} \neq e^{\hat{\omega}_1 + \hat{\omega}_2} \) in general.

**Logarithms of SO(3)**

Can every element in \( SO(3) \) be written in an exponential form? The answer is yes, because we can constructively find an element that exponentiates to a given element in \( SO(3) \).
Proposition 2 (Logarithms of SO(3)) For any $R \in \text{SO}(3)$, there exists a (not necessarily unique) $\omega \in \mathbb{R}^3$ such that

$$R = e^{\hat{\omega}}.$$

We denote the inverse of the exponential map by $\hat{\omega} = \ln(R)$. The corresponding $\omega$ is given by

$$\theta = \|\omega\| = \arccos \left( \frac{\text{trace}(R) - 1}{2} \right)$$

$$\frac{\omega}{\theta} = \frac{1}{2 \sin(\theta)} \begin{bmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{bmatrix}.$$

We can also frame the above proposition in an algorithmic language as follows.

**Algorithm 1 Calculation of the Logarithm of an Element in SO(3).**

1: function LOGARITHM($R \in \text{SO}(3)$)  
2: Calculate $\text{trace}(R)$ and $\theta = \arccos\left(\frac{\text{trace}(R) - 1}{2}\right)$.  
3: Calculate 

$$\ln R = \frac{\theta}{2 \sin \theta} (R - R^T),$$

where $\ln R = \omega = \begin{bmatrix} 0 & -w_3 & w_2 \\ w_3 & 0 & -w_1 \\ -w_2 & w_1 & 0 \end{bmatrix} \in \text{so}(3)$.  
4: Apply the vee map to $\omega$ to get 

$$\omega^\vee = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}. $$

return $\ln R \in \text{so}(3)$ and $\omega^\vee \in \mathbb{R}^3$.  
5: end function

Note that when $\theta$ is small, we can use a Taylor expansion to approximate $\sin \theta \approx \theta$. Hence, $\ln R \approx \frac{R - R^T}{2}$, which is exactly the skew symmetric
component of the matrix \( R \).

**Proof** Given \( R \in \text{SO}(3) \), we want to solve for \( \omega \in \text{so}(3) \) such that

\[
R = \exp(\omega).
\]

(2.3)

Applying the Rodrigues’ formula, we have

\[
\exp(\omega) = I + \frac{\sin \theta}{\theta} \omega + \frac{1 - \cos \theta}{\theta^2} \omega^2,
\]

(2.4)

where \( \theta = \|\omega^\vee\| \). We note that \( \text{trace}(\frac{\sin \theta}{\theta} \omega) = 0 \), so

\[
\text{trace}(R) = \text{trace}(I) + \frac{1 - \cos \theta}{\theta^2} \text{trace}(\omega^2)
\]

\[
= 3 + \frac{1 - \cos \theta}{\theta^2} (-2(w_1^2 + w_2^2 + w_3^2))
\]

\[
= 3 + \left(\frac{-2\theta^2}{\theta^2}\right)(1 - \cos \theta) = 1 + 2 \cos \theta.
\]

Thus, \( \theta = \arccos \left(\frac{\text{trace}(R)-1}{2}\right) \). Note that \( \omega \) is skew symmetric, while \( I \) and \( \omega^2 \) are symmetric, so we get from Proposition[1] that

\[
R - R^T = 2 \frac{\sin \theta}{\theta} \omega.
\]

Thus,

\[
\omega = \frac{\theta}{2 \sin \theta} (R - R^T).
\]


### 2.4.2 Rigid-Body Motion

Suppose that the camera is fixed and the object is rigid and moving. To describe a motion of the whole camera, it is sufficient to specify the motion of one point and the motion of three coordinate axes attached to that point.

**Definition 9 (\( \text{E}(3) \))**

\( \text{E}(3) \) is defined as the set of the 3-dimensional rigid transformations.
However, it is not sufficient to characterize a physically possible map by restriction all the maps to rigid transformations. Consider, for example, the reflection. It preserves the distance but does not preserve orientation. It is impossible to move an object in the physical world in a way that the end result has a different orientation from the original object. Thus, we need to add an additional requirement for a map to preserve orientation of a frame as well. Then we can define the special rigid transformation group:

**Definition 10 (SE(3))**

\[ \text{SE}(3) \overset{\text{def}}{=} \text{the set of the 3-dimensional special rigid transformations} \]

where the adjective *special* in this context means that the transformation also preserves orientation and rigid transformations are the map that preserve the distance.

**Exponential Maps**

Consider a map \( g : \mathbb{R} \to \text{SE}(3) \) We can write \( g(t) = (R(t), T(t)) \) or, in a matrix representation,

\[
g(t) = \begin{bmatrix} R(t) & T(t) \\ 0 & 1 \end{bmatrix} \in \mathbb{R}^{4 \times 4}.
\]

Then

\[
\dot{g}(t)g^{-1}(t) = \begin{bmatrix} \dot{R}(t)R^T(t) & \dot{T}(t) - R(t)R^T(t)T(t) \\ 0 & 0 \end{bmatrix}
\]

We can write \( \dot{T}(t) - \dot{R}(t)R^T(t)T(t) = \ddot{T}(t) - \ddot{\omega} T(t) = v(t) \), where \( v(t) \in \mathbb{R}^3 \). So,

\[
\dot{g}(t)g^{-1}(t) = \begin{bmatrix} \ddot{\omega} & v \\ 0 & 0 \end{bmatrix} = \hat{\xi}.
\]

Thus,

\[
\dot{g}(t) = \hat{\xi}g(t),
\]

where a \( 4 \times 4 \) matrix of the form \( \hat{\xi} \) is called a *twist*.

**Definition 11 (se(3))** The set of all twists is defined to be

\[
\text{se}(3) = \left\{ \hat{\xi} = \begin{bmatrix} \ddot{\omega} & v \\ 0 & 0 \end{bmatrix} | \ddot{\omega} \in \text{SO}(3), v \in \mathbb{R}^3 \right\}
\]
Twist coordinates (6 DOF) can be obtained via the vee map:

\[
\begin{bmatrix}
\dot{\omega} & \nu \\
0 & 0
\end{bmatrix} \vee = \begin{bmatrix}
\nu \\
\omega
\end{bmatrix} \in \mathbb{R}^6.
\]

In the twist coordinates \( \xi \in \mathbb{R}^6 \), \( \nu \) is called the linear velocity and \( \omega \) the angular velocity.

Now how do we quickly compute the exponentiation? We can use the Rodrigue’s formula. We have

\[
e^{\xi} = \begin{bmatrix}
e^{\hat{\omega}} & \frac{(1-e^{\hat{\omega}})\hat{\omega}\nu + \omega^T\nu}{\theta} \\
0 & 1
\end{bmatrix}, \quad \theta \neq 0,
\]

where we recall that \( \theta = ||\omega|| \). If \( \theta = 0 \), we simply have \( e^{\xi} = \begin{bmatrix} I & \nu \\
0 & 1 \end{bmatrix} \).

2.4.3 Logarithms of SE(3)

Can every element in SE(3) be written in an exponential form? The answer is yes.

**Proposition 3 (Logarithms of SE(3))** Suppose \( g = \begin{bmatrix} R & T \\
0 & 1 \end{bmatrix} \in SE(3) \).

Then there exists \( \xi = \begin{bmatrix} \nu \\
\omega \end{bmatrix} \in \mathbb{R}^6 \) such that

\[
e^{\xi} = g.
\]

This means that \( e^{\hat{\omega}} = R \). If \( ||\omega|| \neq 0 \), we have

\[
\frac{(I-e^{\hat{\omega}})\hat{\omega}\nu + \omega^T\nu}{\theta} = T, \quad \theta \neq 0.
\]

If \( R = I \), then \( \theta = 0 \), and we can simply choose \( \omega = 0 \) and \( \nu = T \). \(\square\)
2.4.4 Lie bracket

The Lie bracket is defined by

\[
\hat{\xi}_1 \hat{\xi}_2 - \hat{\xi}_2 \hat{\xi}_1 = \begin{bmatrix}
\hat{w}_1 \times \hat{w}_2 & \hat{w}_1 \times \hat{v}_2 - \hat{w}_2 \times \hat{v}_1 \\
0 & 0
\end{bmatrix} \in \text{se}(3).
\]

The linear structure of so(3) together with the Lie bracket form the Lie algebra of the Lie group SO(3). More details about Lie algebra can be found in Section 4.1.

2.5 Coordinate and velocity transformations

Suppose we track a camera:

\[
g(t) = \begin{bmatrix}
R(t) & T(t) \\
0 & 1
\end{bmatrix} \in \text{SE}(3).
\]

Suppose a point \( p \in \mathbb{E}^3 \) is \( X_0 = X(0) \).

Its coordinates relative to the camera at time \( t \) are given by \( X(t) = R(t)X_0 + T(t) \), or in the homogeneous coordinates, \( \tilde{X} = g(t)\tilde{X}_0 \). If the camera is at locations \( g(t_1), g(t_2), \ldots, g(t_m) \) at times \( t_1, t_2, \ldots, t_m \), respectively. When time is not a focus, we can denote \( X_i = X(t_i), g_i = g(t_i), T_i = T(t_i) \).

Then

\[
X_i = R_iX_0 + T_i = g_iX_0.
\]

The relationship between coordinates of the same point \( p \) at different times:

\[
X(t_2) = g(t_2, t_1)X(t_1), \quad \text{for all} \ t_1, t_2 \in \mathbb{R}.
\]

Also, the composition rule and the inverse rule must hold:

\[
g(t_3, t_1) = g(t_3, t_2)g(t_2, t_1)
\]

\[
g(t_2, t_1) = g(t_1, t_2)^{-1}.
\]

for all \( t_1 < t_2 < t_3 \) in \( \mathbb{R} \).

2.5.1 Rules of velocity transformation

Suppose, in the world frame, we have a point

\[
p = X(t) = g_{cw}(t)X_0.
\]
Taking the derivative of both sides, we get
\[ \dot{\hat{X}}(t) = \dot{g}_{cw}(t)X_0. \]

Define \( \hat{V}^c_{cw}(t) = g_{cw}g_{cw}^{-1} \in \text{se}(3) \). Then we have
\[ \dot{\hat{X}}(t) = g_{cw}(t)g_{cw}^{-1}X(t) = \hat{V}_{cw}(t)X(t). \]

Suppose a viewer is in another frame displaced relative to the camera frame by a rigid body transformation \( g \in \text{SE}(3) \). Then, \( Y(t) = gX(t) \). We obtain the velocity
\[ \dot{Y}(t) = g\dot{g}_{cw}(t)g_{cw}^{-1}(t)g^{-1}Y(t) = g\hat{V}_{cw}(t)g^{-1}Y(t). \]

**Definition 12 (Adjoints maps)** The adjoint map \( \text{adj}_g : \text{se}(3) \to \text{se}(3) \) is defined by
\[ \hat{\xi}_g \mapsto g\hat{\xi}_g g^{-1}. \]

Thus, we can see that, if \( \hat{V} = \text{adj}_g(\hat{V}_{cw}(t)) \), then \( \dot{Y}(t) = \hat{V}Y(t) \).
### 2.6 Summary

<table>
<thead>
<tr>
<th></th>
<th>Rotation SO(3)</th>
<th>Rigid-body motion SE(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Matrix representation</strong></td>
<td>$R \in \mathbb{R}^{3 \times 3}$ such that $R^T R = I$, $\det(R) = 1$</td>
<td>$g = \begin{bmatrix} R &amp; T \ 0 &amp; 1 \end{bmatrix}$</td>
</tr>
<tr>
<td><strong>Coordinate (3D)</strong></td>
<td>$X = RX_0$</td>
<td>$X = RX_0 + T$</td>
</tr>
<tr>
<td><strong>Inverse</strong></td>
<td>$R^{-1} = R^T$</td>
<td>$g^{-1} = \begin{bmatrix} R^T &amp; -R^{-1}T \ 0 &amp; 1 \end{bmatrix}$</td>
</tr>
<tr>
<td><strong>Composition</strong></td>
<td>$R_{ik} = R_{ij}R_{jk}$</td>
<td>$g_{ik} = g_{ij}g_{jk}$</td>
</tr>
<tr>
<td><strong>Exp. representation</strong></td>
<td>$R = e^{\hat{\omega}}$</td>
<td>$g = e^{\hat{\xi}}$</td>
</tr>
<tr>
<td><strong>Velocity</strong></td>
<td>$\dot{X} = \hat{\omega}X$</td>
<td>$\dot{g} = \hat{\xi}g$</td>
</tr>
<tr>
<td><strong>Adjoint map</strong></td>
<td>$\hat{\omega} \mapsto R\hat{\omega}R^T$</td>
<td>$\hat{\xi} \mapsto g\hat{\xi}g^{-1}$</td>
</tr>
</tbody>
</table>
Chapter 3

Obstacle Avoidance

In this chapter, we give an overview of various aspects of robot obstacle avoidance problem, ranging from environment mapping, obstacle representation, to several approaches to find a collision-free trajectory.

3.1 Related Work

Schulman et al. [2013] proposed a novel approach called sequential convex optimization to find locally optimal, collision-free trajectories. They used $l_1$ penalties for equality and inequality constraints, and converted globally non-convex optimization into locally convex problems that are sequentially solved. This approach has advantages of speed of computation and reliability to solve a large portion of planning problems.

In what follows, we consider instead a geometric point of view to approach the obstacle avoidance problem. We also explore an algorithm that builds on a finite state machine, and another one builds on a searching algorithm.

3.2 Environment Map Representation

There are several ways to represent a map of the environments. In our case, we use a 2D grid representation, where each point has an associated height value of the position of the first object it hits if we draw a vertical line from the ground. This representation has downsides of not being able to represent surfaces stacked above another surface. However, in our case study, we will first deal with a cylinder-shaped obstacle, and hence this
map representation is sufficient.

### 3.3 Obstacle Representation

In our case, we use an $n \times n$ 2D grid representation of obstacle, where each point has an associated value between 0 and 1, representing the probability that a grid cell contains obstacle. The probability of 1 at grid cell means that there is definitely obstacle at the grid cell, whereas 0 means that there is definitely no obstacle at the grid cell. The following MATLAB code shows this representation of our map and obstacle.

```matlab
% initialize map
n = 100;
map = zeros(n);
height = 20;

% say an obstacle is a circle in the middle
obstacle = zeros(n);
midx = n/2;
midy = n/2;
circle_radius = n/4;
for i = 1:n
    for j = 1:n
        if((i - midx)^2 + (j - midy)^2 <= circle_radius^2)
            obstacle(i, j) = 1;
        end
    end
end
map = map + obstacle;

% create a path
max_time = n;
for t = 1:max_time
    x_naive(t) = t;
y_naive(t) = t;
z_naive(t) = height/2;
end

% visualize the path
scatter3(x_naive, y_naive, z_naive, 20, 'k');

% make it a probability
map = min(1, max(0, map));

% visualize map
```
An example of obstacle visualization can be found in Fig. 3.1.

3.4 Problem Setup

Suppose our obstacle has the shape of a cylinder. This setup will allow us to see how the robot find a curved optimal path. The start point is at \((-a, 0, h)\). The end point is at \((b, 0, h)\), where \(a, b > 1\) and \(h > 0\), and the cylinder is \(\{(x, y, z) \in \mathbb{R}^3 \mid x^2 + y^2 = 1, 0 \leq z \leq h\}\).

Figure 3.1 A cylinder obstacle problem setup. The start point is at \((-a, 0, h)\). The end point is at \((b, 0, h)\), where \(a, b > 1\) and \(h > 0\). The cylinder is \(\{(x, y, z) \in \mathbb{R}^3 \mid x^2 + y^2 = 1, 0 \leq z \leq h\}\).
3.5 Collision Free Path

Because a robot is not a point, how can we design a path that ensures no obstacle avoidance? A well-known technique is the Minkowski sum, which reduces the problem of designing robot motion to a trajectory planning.

3.5.1 Minkowski Sum

Let $A$ and $B$ be connected subsets of $\mathbb{R}^2$. Let the Minkowski sum $\oplus$ of $A$ and $B$ be defined by

$$A \oplus B = \{a + b | a \in A, b \in B\}.$$ 

We can see that, if we can pick an arbitrary fixed point inside the robot and draw a path that does not touch the Minkowski sum of the robot and the obstacle, we will get an obstacle-free robot motion, and vice versa.

Thus, we have reduced the problem to finding a trajectory problem. Figure 3.2 shows an example of the Minkowski sum.

![Figure 3.2 An example of the Minkowski sum.](image)

3.5.2 Trajectory Generation

We need to specify a cost function to minimize. There could be several objectives that we should consider

- Minimizing time,
• Minimizing distance,
• Minimizing waiting time, or
• Minimizing certain energy functions such as the smoothness of a trajectory.

There are also other considerations for an end-to-end system such as safety, energy, planning speed, and probability of a successful flight. The safety consideration often primarily includes obstacle avoidance. Planning speed can be an issue if it takes too long because it can prevent real-time execution. In this thesis, we will focus on obstacle avoidance.

3.5.3 Approaches

Suppose the robot has a radius $r$. Then we inflate the obstacle and get an inflated obstacle of radius $1 + r$. Suppose we want to minimize the length of the trajectory, which is assumed to be a smooth curve $a(t) = (x(t), y(t))$, and without loss of generality we can assume $0 \leq t \leq 1$. Note that the length of the trajectory is given by $l(t) = \int_{t=0}^{1} |a'(t)| dt = \int_{t=0}^{1} \sqrt{(x'(t))^2 + (y'(t))^2} dt$. At first, we can formulate the problem as an optimal solution problem:

$$\min_{a(t) = (x(t), y(t))} \int_{t=0}^{1} \sqrt{(x'(t))^2 + (y'(t))^2} dt$$

subject to

$$x(0) = -a$$
$$x(1) = b$$
$$y(0) = 0$$
$$y(1) = 0$$
$$x^2(t) + y^2(t) \geq (1 + r)^2 \text{ for all } t,$$

where the constraint $x^2(t) + y^2(t) \geq (1 + r)^2$ means that the trajectory is collision-free (using the Minkowski sum). However, it is unclear how to get a closed form solution in this optimization formulation through common techniques. Fortunately, we can use geometry and symmetry of the problem to get an optimal solution.

It can be proved that an optimal length solution is a path that starts from the start point, go straight to the tangent point of the circle, go around
the circle up to the tangent point from the goal from the other side. Then finally it go straight to the goal. This solution is illustrated in Figure 3.4.

A picture of an experiment setup for this obstacle avoidance problem is shown in Figure 3.4. More details about the experiment can be found in Chapter 7.

**Figure 3.3** An optimal length collision-free path.

**Figure 3.4** A drone obstacle avoidance problem.
3.5.4 Hybrid Automata Model

As a starting point to develop an obstacle avoidance algorithm, a hybrid automata model builds off of a finite state machine. A finite state machine is a set of states and transitions between them. The one that we will consider is a deterministic finite automaton, which is a finite state machine that accepts or rejects finite sequences of transitions and only produces a unique computation of the automaton for each input string. In other words, each state together with a transition must correspond exactly to one state (not necessarily different from the current state).

This section will give a high-level idea of how this model can allow the robot to autonomously reach a goal destination without hitting an obstacle. In Fig. (3.6), the point $O$ represents an obstacle. The Start and Goal points are marked. From Start, the robot needs to go to Goal without going too close to $O$. We draw two circles, one with radius $d$, called the dangerous radius, and the other one with $d'$, called the safe radius, where $d < d'$. Our high-level desired behavior is that the robot’s priority is to go to Goal unless the distance between the robot and $O$ is less than the dangerous radius $d$. If that is the case, the robot will change their priority to avoid the obstacle, until it gets back to the safe radius, in which case the robot changes its priority back to the Go to Goal behavior.

To represent this notion, we utilized a hybrid finite automata machine, which is a set of states and conditions to move between those states. In this case, we use two states (represented by the blue state and the orange state). In the blue state, the robot is controlled by

$$\dot{x} = f_{GTG}(x),$$

where $f_{GTG}$ is a control function that represents a Go to Goal behavior. Similarly, in the orange state, the robot is controlled by

$$\dot{x} = f_{AO}(x),$$

where $f_{AO}$ is a control function that represents a Avoid Obstacle behavior. See Fig. (3.6) for the hybrid automata model schema. We will discuss implementations of each of the two functions in later sections.

3.6 Trajectory Search Algorithms

Another approach uses a searching algorithm to find the optimal trajectory. Chaudhari (2011) presented the rapidly expanding random tree algorithm in the context of the quadrotor problem.
Rapidly Expanding Random Tree (RRT*)

The rapidly expanding random tree algorithm is a heuristic algorithm to go to goal without hitting obstacles. See Fig. (3.7).

Chaudhari uses the differential flatness property of quadrotors to convert a trajectory planning to the control and state estimation. For the trajectory planning part, the RRT* algorithm combines with a sampling method a Dijkstra’s algorithm that finds a minimum path in graphs. The pseudocode of the RRT* algorithm is shown in Algorithm 2.

Algorithm 2 The RRT* Algorithm

1: $V \leftarrow \{z_{init}\}; E \leftarrow \emptyset; i < 0$
2: while $i < N$ do
3:    $G \leftarrow (V, E)$
4:    $z_{rand} \leftarrow \text{Sample}(i); i \leftarrow i + 1$
5:    $(V, E) \leftarrow \text{Extend}(G, z_{rand})$
6: end while
The $z_{\text{init}}$ is the initial state. The “Sample” method returns independent and identically distributed samples from the obstacle-free space, and the “Extend” method creates a new graph after including edges from calls of the Steer procedure on all neighbors of $z$.

Chaudhari then uses polynomials approximation to determine local (point-to-point) steering, based on a time-optimal cost function. He uses the symbolic toolbox in MATLAB to implement the RRT* algorithm and translate into C expressions. Various examples are demonstrated how this algorithm performs. One of the examples is a scenario in which there is wall immediately in the front of a quadrotor. In this case, the RRT* algorithm forces the quadrotor is roll on its sideway in order to achieve the time-optimal path.
Chapter 4

Optimal Control of 3D Robot on SE(3)

In this chapter we devote our study to using a theoretical approach to solve an optimal control problem via a Hamiltonian system. In some special cases, we can solve for a closed form solution. In other cases, we might want to use numerical simulation to solve the problem. In this chapter, however, we will focus solely on the former problem. In this chapter, we follow a Hamiltonian approach by Walsh et al. (1994) to solve the optimal control problem.

4.1 Lie Algebra

The goal of this section is to understand Lie algebra and its application on an improvement on calculation speed of rigid body motion transformations. The key idea of fast calculation is to use a locally best approximation of the space SO(3). First, we define Lie algebra.

Lie Algebra: A Lie algebra is a vector space $L$ over a field $F$ together with a binary operator $\{\cdot, \cdot\} : L \times L \rightarrow L$ which satisfies the following properties

- Bilinear:
  
  \[
  [ax + by, z] = a[x, z] + b[y, z],
  \]
  
  \[
  [z, ax + by] = a[z, x] + b[z, y],
  \]

  for all scalars $a, b$ in $F$ and for all $x, y, z \in L$. 

• Alternating on \( L \):
\[
[x, x] = 0, 
\]
for all \( x \in L \).

• The Jacobi identity:
\[
[x, [y, z]] + [y, [z, x]] + [z, [x, y]] = 0, 
\]
for all \( x, y, z \in L \).

**Remark:** For a Lie group, it has matrix representation
\[
[A, B] = AB - BA \quad (A, B \in T_e(G)).
\]

### 4.2 Optimal Control

Consider an \( n \)-dimensional matrix Lie group \( G \). The associated Lie algebra \( \mathfrak{g} \) is the vector space \( T_e G \) together with the Lie bracket \([\cdot, \cdot]\). We identify the tangent bundle \( TG \) with \( G \times \mathfrak{g} \). For an element \( \dot{g} \in T_g G \), the representative element in \( \mathfrak{g} \) is \( \dot{\epsilon} = d(g^{-1}) \dot{g} \). We will always note elements of \( \mathfrak{g} \) in the form \( \dot{\epsilon} \), while elements of the dual space \( \mathfrak{g}^* \) will be denoted \( \dot{p} \). A left-invariant control system with drift is given by
\[
\dot{g} = g \dot{\epsilon}_0 + \sum_{i=1}^{m} u_i g \dot{\epsilon}_i, \tag{4.1}
\]
where the \( u_i \) are control inputs to the system. The cost function to be minimized is
\[
J = \frac{1}{2} \int_0^T u^T(\tau) M u(\tau) \, d\tau, \tag{4.2}
\]
with \( M \in \mathbb{R}^{m \times m} \) positive definite and symmetric. Here \( m \) is the number of control parameters of the system. The system is subject to \( g(0) = g_0 \) and \( g(T) = g_f \).

**Proposition 4 (Maximum Principle)** Trajectories of the control system (4.1) generated by inputs which minimize (normal extremal) the cost described by (4.2) are solutions of the Hamiltonian system with Hamiltonian
\[
H(g, \dot{p}) = P_0 + \frac{1}{2} P^T \epsilon M^{-1} P_\epsilon, \tag{4.3}
\]
where \( P_i = \dot{\hat{p}}(\dot{\hat{e}}_i) \) and \( P_c \) is the vector of momenta \( P_i \) corresponding to the controlled coordinates.

\[ \square \]

**Proof (Sketch)** The control Hamiltonian for normal extremals is written as follows:

\[
H(g, \dot{\hat{p}}, u) = P_0 + u^T P_c - \frac{1}{2} u^T M u. \tag{4.4}
\]

Because our vector fields are left-invariant, the control Hamiltonian is independent of the system state \( g \). Pontryagin’s maximum principle states that the optimal controls \( u_{\text{max}} \) will maximize the control Hamiltonian at every point of \( T^*G \) for fixed \( g, \dot{\hat{p}} \). The control Hamiltonian is a quadratic function of the scalar \( u \), and because we are considering normal extremals, \( -H_{uu}(g, \dot{\hat{p}}) = M > 0 \). The unique maximizing inputs are then given by

\[
u_{\text{max}} = M^{-1} P_c, \tag{4.5}\]

which may also be expressed as

\[
(u_{\text{max}})_i = \sum_{j=1}^{m} \left( M^{-1} \right)_{ij} P_j. \tag{4.6}\]

Recalling that \( M \) is symmetric, the substitution of these controls into the control Hamiltonian yields

\[
H(g, \dot{\hat{p}}, u) = P_0 + u^T P_c - \frac{1}{2} u^T M u
\]

\[
= P_0 + P_c^T M^{-1} P_c - \frac{1}{2} P_c^T M^{-1} M M^{-1} P_c
\]

\[
= P_0 + \frac{1}{2} P_c^T M^{-1} P_c,
\]

completing the proof of Proposition 4. □

Our goal is to determine the optimal control of a given system (4.1) and cost function (4.2), and then to find the equations of motion for the system given that optimal controls are supplied. We find the optimal controls \( u \) using (4.6). To determine the equations of motion, we first compute the Hamiltonian using (4.3). We then determine the equations of motion for the
Optimal Control of 3D Robot on SE(3)

generalized momenta using Hamilton’s equations, which can be written in the form

\[ \dot{P}_i = \{P_i, H\}, \quad (4.7) \]

where \( \{\cdot, \cdot\} \) is the Poisson bracket.

**Definition 13** In canonical coordinates \((q_i, p_j)\), \(i = 1, \ldots, N\) on phase space, given two functions \(f(p_i, q_i, t)\) and \(g(p_i, q_i, t)\), the Poisson bracket is given by

\[ \{f, g\} = \sum_{i=1}^{N} \left( \frac{\partial f}{\partial q_i} \frac{\partial g}{\partial p_i} - \frac{\partial f}{\partial p_i} \frac{\partial g}{\partial q_i} \right) \quad (4.8) \]

For our purposes, we make use of the convenient identity \( \{P_i, P_j\} = -\hat{p}([\hat{e}_i, \hat{e}_j]) \), where \([\cdot, \cdot]\) is the Lie bracket.

### 4.3 Example Systems

#### 4.3.1 SE(2) Planar Elastica (The Robot Car Problem)

Consider an optimal control problem for a Hilare-like robot car. In this scenario, the car always drives forward at a fixed velocity, and we can control its steering. Note that “elastica” problems refer to those that fix a starting point or end point or both. The problem statement is, given

- an initial position and initial orientation,
- a final goal and orientation,
- a fixed time,

we want to find the optimal steering control with respect to the cost function \( (4.2) \).

Suppose we represent a state in SE(2) by a \( 3 \times 3 \) matrix

\[ g = \begin{bmatrix} R & x \\ 0 & 1 \end{bmatrix}, \]

where \( R \in \text{SO}(2) \), a \( 2 \times 2 \) rotation matrix, and \( x \in \mathbb{R}^2 \).

Suppose the \( \hat{e}_i \) where \( i \in \{1, 2, 3\} \) form the basis set for the Lie algebra se(2), which can be viewed as the tangent space of SE(2) at the identity. We have

\[ \hat{e}_1 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \hat{e}_2 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \quad \hat{e}_3 = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}. \]
We can see that $\hat{e}_1$ and $\hat{e}_2$ correspond to the translations in the $x$ direction and $y$ direction, respectively, and $\hat{e}_3$ corresponds to rotation counterclockwise. Thus, we see the dynamics of our system are

$$\dot{g} = g\hat{e}_3u_3 + g\hat{e}_1,$$

where $u = [0, 0, u_3]$ is the control function and $\hat{e}_0 = \hat{e}_1$ is the drift term, since we can only control the rotation and we assume that the car always drive forward. The cost of a trajectory is given by (4.2) with $M = [1]$. That is,

$$J = \frac{1}{2} \int_0^T u_3^2 \, d\tau. \quad (4.9)$$

From Proposition 4, we get that

$$u_{\text{max}} = [u_3]_{\text{max}} = M^{-1}\hat{p} = [P_3].$$

Therefore, the optimal input is $u_3 = P_3$. From Proposition 4 we compute the Hamiltonian

$$H(g, \hat{p}) = P_1 + \frac{1}{2}P_3^2.$$

We then use the Poisson bracket to get the dynamics of the system:

$$\dot{P}_i = \{P_i, H\},$$

for $i \in \{1, 2, 3\}$. We note that,

$$\{P_i, P_j\} = -\hat{p}(\hat{e}_i, \hat{e}_j),$$

where $[\cdot, \cdot]$ is the Lie bracket. For example, we can get

$$\{P_1, P_3\} = -\hat{p}(\hat{e}_1, \hat{e}_3)$$

$$= -\hat{p}(\hat{e}_1\hat{e}_3 - \hat{e}_3\hat{e}_1)$$

$$= -\hat{p} \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} - \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$= -\hat{p} \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

$$= \hat{p}(\hat{e}_2)$$

$$= P_2.$$

Following this kind of computation, we get the following Poisson bracket table
Thus, we get that
\[
\dot{P}_1 = \{P_1, H\} = \{P_1, P_1\} + \frac{1}{2}\{P_1, P_3^2\} = 0 + P_3\{P_1, P_3\} = P_2P_3,
\]
where we have used the fact that \(\{x, x\} = 0\) and \(\{x, yz\} = y\{x, z\} + z\{x, y\}\), for all \(x, y, z\). In other words, \(\{x, y^2\} = 2y\{x, y\}\).

Using the Poisson bracket table and the Hamiltonian, we get that the dynamics of \(P\) are
\[
\dot{P}_1 = P_2P_3 \\
\dot{P}_2 = -P_1P_3 \\
\dot{P}_3 = -P_2.
\]

### 4.3.2 SE(3) Planar Elastica (The UAV Problem)

We now consider the airplane problem, in which we wish to steer a kinematic airplane from some initial position, orientation, and time to an assigned final position, orientation, and time, using the 6 dimensional configuration space \(SE(3)\).

We represent a state in \(SE(3)\) by a \(4 \times 4\) matrix
\[
\mathbf{s} = \begin{bmatrix} R & x \\ 0 & 1 \end{bmatrix},
\]
where \(R \in SO(3)\), a \(3 \times 3\) rotation matrix, and \(x \in \mathbb{R}^3\).

Suppose the \(\hat{\mathbf{e}}_i\), where \(i \in \{1, 2, 3, 4, 5, 6\}\) form the basis set for the Lie algebra \(se(3)\), which can be viewed as the tangent space of \(SE(3)\) at the identity. We have
\[
\begin{align*}
\hat{\mathbf{e}}_1 &= \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \\
\hat{\mathbf{e}}_2 &= \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \\
\hat{\mathbf{e}}_3 &= \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \\
\hat{\mathbf{e}}_4 &= \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \\
\hat{\mathbf{e}}_5 &= \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \\
\hat{\mathbf{e}}_6 &= \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}.
\]
We can see that $\hat{e}_1$, $\hat{e}_2$, and $\hat{e}_3$ correspond to counterclockwise rotations about the $x$, $y$, and $z$ axes, respectively, and $\hat{e}_4$, $\hat{e}_5$, and $\hat{e}_6$ correspond to translation along these directions. Thus, we see the dynamics of our system are

$$\dot{g} = g\hat{e}_1 u_1 + g\hat{e}_2 u_2 + g\hat{e}_3 u_3 + g\hat{e}_4,$$

where $u = [u_1, u_2, u_3, 0, 0, 0]$ is the control function and $\hat{e}_0 = \hat{e}_4$ is the drift term. The cost of a trajectory is given by

$$J = \frac{1}{2} \int_0^T \sum_{i=1}^3 c_i u_i^2(\tau) \, d\tau. \quad (4.10)$$

In this case

$$M = \begin{bmatrix} c_1 & 0 & 0 \\ 0 & c_2 & 0 \\ 0 & 0 & c_3 \end{bmatrix}$$

From Proposition 4, we see that

$$u_{\text{max}} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = M^{-1} P_c = \begin{bmatrix} P_1 / c_1 \\ P_2 / c_2 \\ P_3 / c_3 \end{bmatrix}.$$ 

Therefore, the optimal inputs are $u_1 = P_1 / c_1$, $u_2 = P_2 / c_2$, $u_3 = P_3 / c_3$.

From Proposition 4 we compute the Hamiltonian

$$H(g, p) = P_4 + \frac{1}{2} \left( \frac{P_1^2}{c_1} + \frac{P_2^2}{c_2} + \frac{P_3^2}{c_3} \right).$$

We then use the Poisson bracket to get the dynamics of the system, as before:

$$\dot{P}_i = \{P_i, H\},$$

for $i \in \{1, 2, 3, 4, 5, 6\}$. In this case, we get the following Poisson bracket table

<table>
<thead>
<tr>
<th></th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
<th>$P_5$</th>
<th>$P_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0</td>
<td>$-P_3$</td>
<td>$P_2$</td>
<td>0</td>
<td>$-P_6$</td>
<td>$P_5$</td>
</tr>
<tr>
<td>$P_2$</td>
<td>$P_3$</td>
<td>0</td>
<td>$-P_1$</td>
<td>$P_6$</td>
<td>0</td>
<td>$-P_4$</td>
</tr>
<tr>
<td>$P_3$</td>
<td>$-P_2$</td>
<td>$P_1$</td>
<td>0</td>
<td>$-P_5$</td>
<td>$P_4$</td>
<td>0</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0</td>
<td>$-P_6$</td>
<td>$P_5$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$P_5$</td>
<td>$P_6$</td>
<td>0</td>
<td>$-P_4$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$P_6$</td>
<td>$-P_5$</td>
<td>$P_4$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Using the Poisson bracket table and the Hamiltonian, we get that the dynamics of $P$ are

$$
\begin{bmatrix}
\dot{P}_1 \\
\dot{P}_2 \\
\dot{P}_3 \\
\dot{P}_4 \\
\dot{P}_5 \\
\dot{P}_6
\end{bmatrix} =
\begin{bmatrix}
\frac{c_2-c_3}{c_2c_3} P_2 P_3 \\
\frac{c_1-c_3}{c_1c_3} P_1 P_3 \\
-2P_3 + \frac{c_1-c_2}{c_1c_2} P_1 P_2 \\
0 & P_6 & -P_5 \\
-P_6 & 0 & P_4 \\
P_5 & -P_4 & 0
\end{bmatrix}
\begin{bmatrix}
P_1 \\
P_2 \\
P_3 \\
P_4 \\
P_5 \\
P_6
\end{bmatrix}.
$$

(4.11)

**The Lagrange Top**

In the special case of $c_1 = c_2 = c_3 = c$, the resulting system is integrable and is called the Lagrange top. In such a system, we get additional constants of motion, called casimiers,

$$
k_1(g, P) = p_2^2 + p_3^2 + p_6^2,
$$

$$
k_2(g, P) = p_1 p_4 + p_2 p_5 + p_3 p_6,
$$

$$
k_3(g, P) = p_1.
$$

**Elastic Trajectories**

Given a Hamiltonian system $H$ on SE(3) described by (4.11) with cost constants $c_1 = c_2 = c_3 = c$ and with casimiers $k_1, k_2, k_3$. Then, the optimal inputs are given by $u_1(t) = \frac{1}{c} k_3, u_2(t) = \frac{1}{c} r(t) \cos(\vartheta(t))$, and $u_3 = \frac{1}{c} r(t) \sin(\vartheta(t))$, where $r(t)$ and $\vartheta(t)$ satisfy the following

$$
r(t) = \sqrt{2H - k_3^2 - 2P_4(t)},
$$

$$
\dot{\vartheta}(t) = -\frac{k_1 - k_3 P_4(t)}{2H - k_3^2 - 2P_4(t)},
$$

with $P_4(t)$ being an affine transformation of the Weierstrass $P$ function:

$$
P_4(t) = 2^{1/3} P + \frac{1}{3} H.
$$

For a proof of the above statement, see [Walsh et al. 1994].
4.3.3 Inverse Solution

Given a desired location \( x_d, y_d, z_d \) with \( x_d \neq 0 \) and \( y_d^2 + z_d^2 \neq 0 \), the optimal inputs which drive the airplane to the final desired position are

\[
\begin{align*}
    u_1 &= 0, \\
    u_2 &= \rho \sin(\xi), \\
    u_3 &= \rho \cos(\xi), \\
    T &= \frac{\psi}{\rho},
\end{align*}
\]

where

\[
\begin{align*}
    \rho &= \frac{1}{x} \sin \psi, \\
    \psi &= 2 \arctan(\sqrt{y^2 + z^2}, x), \\
    \xi &= 2 \arctan(\sqrt{y^2 + z^2}, x).
\end{align*}
\]

This statement above is a direct application of Rodrigues’ formula, which is

\[
\exp(\hat{\omega}) = I + \frac{\sin \theta}{\theta} \hat{\omega} + \frac{1 - \cos \theta}{\theta^2} \hat{\omega}^2, \tag{4.12}
\]

where \( \theta = \|\omega\| \) and \( \omega \in \mathbb{R}^3 \).

Remark. Because we have exact formulas for the forward problem, we can use numerical methods to perturb the exact inverses above toward the desired solutions very rapidly. Note that, in general, it may be the case that no solution exists for given initial and final states; for example, the system may simply not have enough time to reach its destination.
Chapter 5

Learning for Control from Multiple Demonstrations

Sometimes it is not easy to pre-define a trajectory for a robot to follow, because such a trajectory has to conform with the dynamics of the robot and thus require an accurate dynamics model. Coates et al. (2008) have proposed a different approach to tackle with this problem by incorporating machine learning techniques. They write an algorithm that extracts the initially unknown desired trajectory from the sub-optimal expert’s demonstrations and learns a local model that is suitable for control along the learned trajectory. They apply the algorithm and perform an experiment with the problem of autonomous helicopter.

5.1 Motivation

5.1.1 Problem

Many problems in robots control boil down to the problem of describing a trajectory that a robot should follow. Coates et al. (2008) focus on the autonomous helicopter flight. Designing a desired path that follows the dynamics of the robot is not a trivial task, because we need to have an accurate helicopter dynamics model. This non-linear problem is due to many complicating factors, such as air flow near the helicopter.
5.1.2 Solution: Apprenticeship learning

The problem of autonomous helicopter flight is well-known to be a very challenging problem. Fortunately, human experts can reliably fly helicopters in a wide range of maneuvers. However, this proposed solution raises a new problem.

5.1.3 New Problem

It is not easy to perform a demonstration. Even though the human pilot can have a desired path in mind, it is not feasible to fly the helicopter to exactly follow the path. Further, the sensors are not sufficiently fine-grained to give a smooth path.

5.1.4 New Solution: Learning for trajectory control from multiple demonstrations

In this section, we will infer the “hidden” state and control, treating each expert demonstration as a noisy observation of the optimal hidden state and control. This approach reduces the noise from air flow and other factors so that our final path is clean and follows the helicopter dynamics. The beginning of this chapter is an exploration of work of Coates et al, and we introduce how to incorporate geometry at the end of the chapter.

5.2 Generative Model

A summary of the enhanced generative model is given by

hidden target trajectory: \[ z_{t+1} = f(z_t) + \beta_t^* + \omega^z_t \] (5.1)

a bias term: \[ \beta_{t+1}^* = \beta_t^* + \omega^\beta_t \] (5.2)

a drift vector: \[ \delta_{j+1}^k = \delta_j^k + \omega^{\delta}_j \] (5.3)

prior knowledge: \[ \rho_t = \rho(x_t) + \omega^\rho_t \] (5.4)

an observation: \[ y^k_j = z^k_{\tau^k_j} + \delta^k_j + \omega^y_j \] (5.5)

the time index: \[ \tau^k_j \sim P \left( \tau_{j+1}^k | \tau^k_j \right), \] (5.6)
where $\omega_i^{(z)}$ and $\omega_j^{(z)}$ are zero mean Gaussian random variables with respective covariance matrices $\Sigma^{(z)}$ and

$$P(\tau_{j+1}^k | \tau_j^k) = \begin{cases} d_1^k, & \tau_{j+1}^k - \tau_j^k = 1, \\ d_2^k, & \tau_{j+1}^k - \tau_j^k = 2, \\ d_3^k, & \tau_{j+1}^k - \tau_j^k = 3 \\ 0, & \text{otherwise,} \end{cases} \quad (5.7)$$

where $d_1^k \geq d_2^k \geq d_3^k \geq 0$ are parameters, collectively denoted as $d$. The Stanford model does not impose the condition $d_1^k \geq d_2^k \geq d_3^k$, but we think it makes the most sense to only allow the smaller shift in time to occur with equal or larger probability of the larger shift in time. We also have an initial condition

$$\tau_0^k \equiv 0. \quad (5.8)$$

### 5.3 Explanation

The generative model for the hidden, intended trajectory is given by an initial state distribution $z_0 \sim N(\mu_0, \Sigma_0)$ and an approximate model of the dynamics

$$z_{t+1} = f(z_t) + \omega_t^{(z)}, \quad \omega_t^{(z)} \sim N(0, \Sigma^{(z)}). \quad (5.9)$$

The dynamics model $f$ is obtained by

$$\begin{align*}
\dot{u} &= vr - wq + A_x u + g_x + \omega_t^{(u)}, \\
\dot{v} &= wp - ur + A_y v + g_y + D_0 + \omega_t^{(v)}, \\
\dot{w} &= uq - vp + A_z w + g_z + C_4 u + D_4 + \omega_t^{(w)}, \\
\dot{p} &= qr (I_{yy} - I_{zz}) / I_{xx} + B_x p + C_1 u + D_1 + \omega_t^{(p)}, \\
\dot{q} &= pr (I_{zz} - I_{xx}) / I_{yy} + B_y q + C_2 u + D_2 + \omega_t^{(q)}, \\
\dot{r} &= pq (I_{xx} - I_{yy}) / I_{zz} + B_z r + C_3 u + D_3 + \omega_t^{(r)},
\end{align*}$$

where $(u, v, w)$, $(p, q, r)$, and $(g_x, g_y, g_z)$ denote linear velocities, angular rates, and gravity in the body frame of the helicopter. Note that in the Stanford model, the body coordinates $x, y, z$ correspond to forward, right, and down, respectively, rather than the conventional choice.

The coefficients are fitted by data using linear regression, which minimizes the square error.
In practice, the Stanford team precomputes the dynamics model \( f \) from a large dataset and uses this generic \( f \) for all the performed trajectories.

The generative model represents each demonstration as a set of independent observation of the hidden, intended trajectory \( z \). In other words, the Stanford model assumes

\[
y_j^k = z_{\tau_j^k} + \omega^{(y)}(y), \quad \omega^{(y)} \sim N(0, \Sigma^{(y)}),
\]

(5.16)

where \( \tau_j^k \) is the time index mapping from \( z \) to \( y \). The time index \( \tau_j^k \) are unobserved, and the Stanford model assumes the distribution to follow Equations (5.7) and (5.8), allowing small, gradual shifts in time between hidden and observed trajectories. It is noted that the value of \( T \) that yields a sufficient resolution is equal to twice the average length of the demonstrations. That is,

\[
T = 2 \left( \frac{1}{M} \sum_{k=1}^{M} N^k \right),
\]

(5.17)

gives sufficient resolution.

### 5.3.1 Extensions to the Generative Model

The Stanford model not only accounts for the time alignment, but it takes other important sources of error into account as well. They substantially improve the model by using a time-varying model \( \hat{f} \) that is specific to the vicinity of the intended trajectory:

\[
z_{t+1} = \hat{f}(z_t) + \omega_t^{(z)} = f(z_t) + \beta_t^* + \omega_t^{(z)},
\]

(5.18)

where \( \beta_t^* \sim N(\beta_t^*, \Sigma(\beta)) \).

They further improve the model by including a similar drift term \( \delta_j^k \) that accounts for drift in the demonstrations. Then

\[
y_j^k = z_{y_j^k} + \delta_j^k + \omega_j^{(y)}.
\]

(5.19)

The model also includes the ability to incorporate prior knowledge about the hidden path. This comes in the form of additional observations \( \rho_t = \rho(z_t) \). The function \( \rho \) computes features of the hidden state \( z_t \) and then the expert supplies the value \( \rho_t \) that these features should take. We also assume that these observations may contain Gaussian noise.
5.3.2 Model Schematic

The following model schematic shows all the possible mappings between the observed and hidden trajectories, allowing small, gradual shifts in time between them. The double circle symbol represents an observed quantity, whereas the single circle represents an unobserved quantity.

The dependencies add a layer of complexity to the computation of maximum joint point probability. We shall see that the Stanford model proposes to solve this problem by alternatively fixing a variable (such as $\tau$) and optimizing over other variables.

5.3.3 Model Summary

The generative model is given by

- hidden target trajectory: $z_{t+1} = f(z_t) + \beta_t^z + w_t^{(z)}$
- a bias term: $\beta_{t+1}^z = \beta_t^z + \omega_t^{(\beta)}$
- a drift vector: $\delta_{t+1}^z = \delta_t^z + \omega_t^{(\beta)}$
- prior knowledge: $\rho_t = \rho(x_t) + \omega_t^{(\rho)}$
- an observation: $y_j^k = z_{j+1}^i + \delta_j^k + \omega_j^{(y)}$
- the time index: $\tau_{j}^k \sim P(\tau_{j+1}^k | \tau_j^k)$,

where $\omega_t^{(\cdot)}$ is a zero mean Gaussian random variable with respective covariance matrix $\Sigma^{(\cdot)}$ and

$$P(\tau_{j+1}^k | \tau_j^k) = \begin{cases} a^k, & \tau_{j+1}^k - \tau_j^k = 1, \\ a^k, & \tau_{j+1}^k - \tau_j^k = 2, \\ a^k, & \tau_{j+1}^k - \tau_j^k = 3 \\ 0, & \text{otherwise}, \end{cases}$$
with \( d^k_1 \geq d^k_2 \geq d^k_3 \geq 0 \) are parameters, collectively denoted as \( d \).

Then [Coates et al. (2008)] use the maximum joint likelihood principle to automatically find the time alignment \( \tau \), the time index probabilities \( d \), and the covariance matrix \( \Sigma^{(i)} \). That is, those are parameters that solve

\[
\max_{\tau, \Sigma^{(i)}, d} \log P(y, \rho, \tau; \Sigma^{(i)}, d).
\]

### 5.4 Trajectory Learning Algorithm

Their learning algorithm automatically finds the time alignment indices \( \tau \), the time-index transition probabilities \( d \), and the covariance matrices \( \Sigma^{(i)} \) approximately by maximizing the joint likelihood of the observed trajectories \( y \) and the observed trajectory \( \rho \). As part of the EM algorithm, they marginalize over the unobserved, intended trajectory \( z \).

The optimization problem that we aim to solve is

\[
\max_{\tau, \Sigma^{(i)}, d} \log P \left( y, \rho, \tau; \Sigma^{(i)}, d \right). \tag{5.20}
\]

One way to optimize Equation (5.20) is to alternatively optimize over \( \Sigma^{(i)} \), \( d \), and \( \tau \).

Below is a rough outline of the EM (Expectation-Maximization) algorithm.

**Problem:** The joint optimization (5.20) is very difficult.

**Solution:** Alternately fix one of the variables. Fix \( \tau \).

- \( d \) can be computed in a closed form.
- \( \Sigma^{(i)} \) can be computed using the standard Hidden Markov Model (HMM) parameter learning problem in [Dempster et al. (1977)].

Fix \( \Sigma^{(i)}, d, z \). We can compute the optimal \( \tau \) using the dynamic programming algorithm (a.k.a. dynamic time warping).
Here we provide steps to compute the algorithm. We use the EM (Expectation-Maximization) algorithm.

1. Initialize parameters to some default values. A typical choice is $\Sigma^{(i)} = I, d_i^k = \frac{1}{3}, \tau_j^k = \lceil \frac{j}{N-k-1} \rceil$.

2. E-step for latent trajectory: For the current setting of $\tau, \Sigma^{(i)}$, run an extended Kalman smoother to find the distributions for the latent states $N(\mu_{t|T-1}, \Sigma_{t|T-1})$.

3. M-step for latent trajectory: Update the covariances $\Sigma^{(i)}$ using the standard EM update.

4. E-step for the time indexing: run dynamic time warping to find $\tau$ that maximizes the joint probability $P(y, \rho, \tau; \Sigma, d, H, K)$, where $\bar{z}$ is fixed to $\mu_{t|T-1}$, which is the mode of the distribution obtained from the Kalman smoother.

5. M-step for the time indexing: estimate $d$ from $\tau$.

6. Repeat steps 2-5 until the algorithm converges.

### 5.5 Local Model Learning

After we have time aligned demonstration data, we can build models for state at time $t$: One way is to use locally weighted linear regression (see Atkeson et al. (1997)).

### 5.6 Model Improvement

We plan to include the geometric properties of the curve, such as inflection points $H$ and the curvature $K$, to improve the time alignment step. The joint probability problem then becomes

$$\max_{\tau, \Sigma^{(i)}, d} \log P \left( y, \rho, \tau; \Sigma^{(i)}, d, H, K \right). \quad (5.21)$$

Details on using the curvature $K$ can be found in the next section.
Chapter 6

Theories Meet Practices

We have seen two approaches to solve the optimal control problem: one from the theoretical point of view via a Hamiltonian system, one from a machine learning perspective using human demonstrations.

In this chapter, we continue thinking about what aspects are still missing to make a successful obstacle-free trajectory. We first define the curvature and then explore how to incorporate sensor data to improve the state estimation.

6.1 Improvement on Curve Matching Algorithm

First, we would like to use differential geometry ideas to enhance the existing reinforcement model. The main idea is to use the curvature as features to match.

**Definition 14 (Curvature)** Given an interval $I$ of real numbers, let $\alpha : I \rightarrow \mathbb{R}^3$ be a regular parametrized curve. The curvature of $\alpha$ at $t \in I$ is

$$k(t) = \frac{|\alpha' \times \alpha''|}{|\alpha'|^3}.$$}

Given trajectory data, which are many discrete points, we can use a polynomial curve to fit each window of data, and then we can infer the curvature of each data point. To do this, given a curve

$$\alpha(t) = (x(t), y(t), z(t)),$$

where $t \in I$ for some interval $I$, we find polynomials $p_x(t)$ that best fits $x(t)$ in the interval $I$, $p_y(t)$ that best fits $y(t)$ in the interval $I$, and $p_z(t)$ that best
fits $z(t)$ in the interval $I$. Then our approximated polynomial curve is
\[
\bar{r}(t) = (p_x(t), p_y(t), p_z(t)).
\] (6.1)

In MATLAB, we use a polynomial curve fitting can be done using the following code (in this particular example, we use a polynomial of degree 7):

```matlab
1 t = 1:(max_index - min_index + 1); % time indices
2 xVec = posData(min_index:max_index, 1); % x values
3 yVec = posData(min_index:max_index, 2); % y values
4 zVec = posData(min_index:max_index, 3); % z values
5
6 [fitX, gofX] = fit(t', xVec, 'poly7');
7 [fitY, gofY] = fit(t', yVec, 'poly7');
8 [fitZ, gofZ] = fit(t', zVec, 'poly7');
9 xCoeff = coeffvalues(fitX);
10 yCoeff = coeffvalues(fitY);
11 zCoeff = coeffvalues(fitZ);
12
13 Xvalue = xCoeff * [t.^7; t.^6; t.^5; t.^4; t.^3; t.^2; t; ones(1, length(t))];
14 Yvalue = yCoeff * [t.^7; t.^6; t.^5; t.^4; t.^3; t.^2; t; ones(1, length(t))];
15 Zvalue = zCoeff * [t.^7; t.^6; t.^5; t.^4; t.^3; t.^2; t; ones(1, length(t))];
```

An example of the curve fitting is shown in Figure 6.1.

However, due to the time limit, the question of how to use the information about the curvature to solve (5.21) is still an open problem.

### 6.2 Incorporating Sensor Measurements

As a remark of another aspect of what is missing so far but necessary to make a quadrotor fly successfully, we overview the question of how to incorporate sensor data in order to get a more accurate state estimation. First, we followed the treatment by [Thrun et al., 2001] to consider the Bayes Filter, the Kalman Filter, and then the Extended Kalman Filter. Note that the Extended Kalman Filter is implemented in the code baseline developed by [Engel et al., 2014] that we will use as a starting point for our AR.Drone 2.0 obstacle avoidance experiments.

Consider a discrete time stochastic process. We represent the estimated state (belief) by a Gaussian
\[
x_t \in \mathcal{N}(\mu_t, \Sigma_t).
\]
Figure 6.1  An example of polynomial curve fitting onto data. The black curve is the best polynomial curve of degree 7 that fits the data.
We assume that the system evolves linearly over time, depends linearly on the controls, and has zero-mean, normally distributed process noise

\[ x_t = Ax_{t-1} + Bu_t + \epsilon_t, \]

where \( u \) is the control and \( \epsilon_t \sim \mathcal{N}(0, Q) \). We also assume that observations depend linearly on the state and are perturbed by zero-mean, normally distributed observation noise

\[ z_t = Cx_y + \delta_t \]

with \( \delta \sim \mathcal{N}(0, R) \).

Then we use a belief update. The initial belief is Gaussian

\[ \text{Bel}(x_0) = \mathcal{N}(x_0; \mu_0, \Sigma_0). \]

The next state is also Gaussian, because it is a linear transformation of the previous state:

\[ x_t \sim \mathcal{N}(Ax_{t-1} + Bu_t, Q). \]

Observations are also Gaussian \( z_t \sim \mathcal{N}(Cx_t, R) \)

We recall the properties of normal distribution. If \( X \sim \mathcal{N}(\mu, \Sigma) \) and \( Y \sim AX + B \), then \( Y \sim \mathcal{N}(A\mu + B, A\Sigma A^T) \).

If \( X_1 \sim \mathcal{N}(\mu_1, \Sigma_1) \) and \( X_2 \sim \mathcal{N}(\mu_2, \Sigma_2) \), then

\[
p(X_1, X_2) \sim \mathcal{N}\left(\frac{\Sigma_2}{\Sigma_1 + \Sigma_2} \mu_1 + \frac{\Sigma_1}{\Sigma_1 + \Sigma_2} \mu_2, \frac{1}{\Sigma_1^{-1} + \Sigma_2^{-1}}\right).
\]

For the Kalman Filter, for each step, do

(a) Apply motion model

\[
\text{Bel}(x_t) = \int \frac{p(x_t|x_{t-1}, \mu_t) \text{Bel}(x_t)}{\mathcal{N}(x_t; Ax_{t-1} + Bu_t, Q) \mathcal{N}(x_t; \mu_t, \Sigma_t)} \, dx_{t-1}
\]

\[
= \mathcal{N}(x_t; A\mu_{t-1} + B\mu_t; A\Sigma A^T + Q)
\]

\[
=: \mathcal{N}(x_t; \mu_t, \Sigma_t)
\]

(b) Apply sensor model

\[
\text{Bel}(x_t) = \frac{p(z_t|x_t) \text{Bel}(x_t)}{\mathcal{N}(z_t; Cx_t, R) \mathcal{N}(x_t; \mu_t, \Sigma_t)}
\]

\[
= \mathcal{N}(x_t; \mu_t + K_t(x_t - C\mu_t), (I - K_tX)\Sigma_t)
\]

\[
=: \mathcal{N}(x_t; \mu_t, \Sigma_t),
\]

where \( \eta \) is a constant and \( K_t = \Sigma_t C^T (C\Sigma_T C^T + R)^{-1} \).
Note that the Kalman Filter is highly efficient, since its run-time is polynomial: $O(k^{2.376} + n^2)$ where $k$ is the dimension of the sensor measurement and $n$ is the dimension of the state.

However, it is well known that most robotics systems are nonlinear. An idea to solve this problem is to linearize both the motion function and the observation function around the relevant points. This solution is called Extended Kalman Filter, which uses a linear approximation of the function near points of interest instead.
Chapter 7

Results and Experiments with AR.Drone 2.0

In this section, we focus on implementing and testing algorithms on AR Drone 2.0, which is a commercial product but is commonly used in UAV research. We set up a series of challenges to test our ability to control a quadrotor.

We will give an overview of the experiment setup because this setup can shape the scope of our problems. For example, all other things being equal, a quadrotor with an extremely light weight camera but has a very high frame per second (FPS) definitely has an advantage over a quadrotor with a lower quality camera. In our testing environment, we only use a monocular camera, because we found out that it already performs well, using the codebase and computer vision techniques developed by Engel et al. (2014).

7.1 AR.Drone 2.0 Specifications

First, we review technical specifications of AR.Drone 2.0.

7.1.1 Primary Sensors

- Cameras. There are two cameras: the front camera and the bottom camera
  - 720p 30FPS HD front camera
Results and Experiments with AR.Drone 2.0

Figure 7.1 AR.Drone 2.0 from various points of view, and with and without the indoor protective hull.

- H264 encoding base profile
- Wide angle lens: 92° diagonal
- Low latency streaming
- JPEG photo capture
  - 60 FPS vertical QVGA bottom camera for measuring ground speed
- 3 axis accelerometer ± 50mg precision for measuring all external forces acting upon the quadrotor, including gravity
- Range sensors for determining the distance along a ray
- 3 axis gyroscope 2000°/second precision for measuring orientation
- Ultrasound sensors for measuring ground altitude
- 3 axis magnetometer 6° precision
Pressure sensor ± 10 Pa precision

The on-board electronics on AR.Drone 2 include 1GHz 32 bit ARM Cortex A8 processor with 800MHz video DSP, TMS320DMC64x, Linux 2.6.32, 1GB DDR2 RAM at 200MHz and Wi-Fi b g n. More specifications about the AR.Drone 2 structure and motors can be found in the official specifications.

A monocular camera has advantage of its light weight, and it is already built-in in the drone. A downside is that it cannot directly infer the scale and depth of the image from a few frames.

7.2 Up and Autonomously Flying AR.Drone

Appendix A includes how we set up the system, hardware and software, in the hope that students continuing this project can successfully set up a working end-to-end framework from the group up for developing algorithms for quadrotors. In this section, we give an overview of an important features of our experiment setup.

7.2.1 Hardware

- Parrot AR.Drone 2.0
- A laptop, preferably using Ubuntu 12.04 (Precise), with Wi-fi.
- If a laptop does not have Wi-fi, we can use a Wi-fi USB Connector.
- (optional) USB Joystick or control pad (Linux compatible)

7.2.2 Legacy Navigation Data

Information received from the drone will be published to the `ardrone/navdata` topic. The message type is `ardrone_autonomy::Navdata` and contains the following information:

```c
header: ROS message header
batteryPercent: The remaining charge of the drone’s battery (%)
state: The Drone’s current state:
  * 0: Unknown
  * 1: Inited
```

### Sample NAV Data

To get Navdata (navigation data), we run

```
$ rostopic echo /ardrone/navdata
```

The navdata message is published by the AR.Drone driver at a rate of 50Hz. Sample data recorded at each time step is as follows:

```
header:
  seq: 9221
  stamp:
    secs: 1397661748
    nsecs: 111146320
  frame_id: ardrone_base_link
  batteryPercent: 37.0
  state: 0
  magX: 3
  magY: 15
  magZ: -85
  pressure: 96498
```
temp: 418
wind_speed: 0.0
wind_angle: 0.0
wind_comp_angle: 0.0
rotX: -1.08599996567
rotY: 1.22099995613
rotZ: -18.3159999847
altd: 0
vx: 0.0
vy: -0.0
vz: -0.0
ax: 0.00997698865831
ay: -0.016008593142
az: 0.957161605358
motor1: 0
motor2: 0
motor3: 0
motor4: 0		
tags_count: 0
tags_type: []
tags_xc: []
tags_yc: []
tags_width: []
tags_height: []
tags_orientation: []
tags_distance: []
tm: 735708096.0

In this case, we can also see the tag data, which would help the robot to localize and navigate.

First, we need to connect to the Wi-fi provided by the drone before running these instructions. The Wi-fi from the drone usually has name ardrone_#####.

```bash
# run roscore
roscore

# in another tab, run driver
rosrun ardrone_autonomy ardrone_driver

# in another tab, run stateestimation node
rosrun tum_ardrone drone_stateestimation

# in another tab, run autopilot node
rosrun tum_ardrone drone_autopilot

# in another tab, run gui node
rosrun tum_ardrone drone_gui
```
To use joystick, we also run the following command in another tab:

```
rosrun joy joy_node
```

The graphical user interface (GUI) window allows four modes of control: no control, keyboard control, joystick control, and the autopilot control. We found that a strategy to develop an autonomy flight algorithm is to have a joystick be able to take over the autopilot during the test. We use it for safety reasons when the drone autonomous mode does not behave as expected and also to help the drone reach the desired starting position. In the future, we would try to have the drone operate as much autonomously as possible.

We tested joystick control, and found that we can control the drone to move left, right, forward, backward, spin left, spin right, upwards, and downwards with ease. The take-off (FLY) and land (LAND) commands are executed successfully. The flip (FLIP) command also works, but it is difficult for the robot to stabilize itself after the flip. The flip flag is set to disabled whenever the battery is below 30%.

It is also possible to update the flight planning text file, which resides at the `tum_ardrone/flightPlans` folder. Be sure to set the reference frame to the current position. For example, to make a house-shaped path, use the following path plan:
autoInit 500 800
setReference $POSE$
setMaxControl 1
setInitialReachDist 0.2
setStayWithinDist 0.5
setStayTime 0

goto -1 0 -0.4 0

goto 1 0 -0.4 0

goto -1 0 0.8 0

goto 1 0 0.8 0

goto 0 0 1.4 0

goto -1 0 0.8 0

goto -1 0 0.4 0

goto 1 0 0.8 0

goto 1 0 -0.4 0

goto 0 0 -0.4 0

land

7.3 Goals

7.3.1 Challenge 1: Get the Robot to Follow a Given Trajectory

In this challenge, we give a trajectory for the quadrotor to follow. This challenge would give us an idea of how well we can control the quadrotor. Below is a list of trajectories that we plan to give to the quadrotor, in the order of the level of difficulty from easy to difficult. It is designed to get a sense of controllability of the quadrotor.

First we run stateestimation node, and the result is shown in Figure 7.3.
Results and Experiments with AR.Drone 2.0

Figure 7.3 Parallel Tracking and Mapping (PTAM) and state estimation. The top, middle, and bottom pictures show the beginning, middle, and ending of the feature point tracking process.
With further tracking of feature points, the point cloud is shown in Figure 7.4.

![Point cloud after feature point tracking.](image)

**Figure 7.4** Point cloud after feature point tracking.

A straight line path on a horizontal plane. This task is done by the following flight plan:

```
autoInit 500 800
setReference $POSE$
setMaxControl 1
setInitialReachDist 0.2
setStayWithinDist 0.5
setStayTime 0
goto x y 0 yaw
land
```

where \(x, y\) are the \(XY\) coordinate with respect to the current position (due to our setting `setReference $POSE$`), and \(yaw\) is the yaw angle change that we would like to make.
A straight line path on a vertical plane. This task is done by the following flight plan:

```
1 autoInit 500 800
2 setReference $POSE$
3 setMaxControl 1
4 setInitialReachDist 0.2
5 setStayWithinDist 0.5
6 setStayTime 0
7 goto 0 y z yaw
8 land
```

where $y$, $z$ are the YZ coordinate with respect to the current position (due to our setting setReference $POSE$), and yaw is the yaw angle change that we would like to make. In this case, we fix the $X$ coordinate, so the drone only moves in a straight line in the YZ plane.

A straight line from point $A$ to point $B$. Suppose we are currently at point $A$. Suppose $B - A = (x, y, z)$. This task is done by the following flight plan:

```
1 autoInit 500 800
2
```
setReference $POSE$
setMaxControl 1
setInitialReachDist 0.2
setStayWithinDist 0.5
setStayTime 0
goto x y z yaw
land

where $x$, $y$, $z$ are the XYZ coordinate with respect to the current position (due to our setting setReference $POSE$), and $yaw$ is the yaw angle change that we would like to make.

**A circle.** A circle path is approximated by a series of way points. The way points can be simply generated by MATLAB using the following code

```
R = 2
i = 0:8
y = R .* sin(2*pi.*i/8)
x = R .* cos(2*pi.*i/8)
[x' y']
```

Then we can use those way points to generate a circular path. For example, the above code yields

```
ans =
2.00000 0.00000
1.41421 1.41421
0.00000 2.00000
-1.41421 1.41421
-2.00000 0.00000
-1.41421 -1.41421
-0.00000 -2.00000
1.41421 -1.41421
2.00000 -0.00000
```

Thus, we can write a flight plan as follows:

```
autoInit 500 800
setReference $POSE$
setMaxControl 1
setInitialReachDist 0.2
setStayWithinDist 0.5
setStayTime 0
```

```
A curve with low curvature. This task is done similarly to the case of a circle. We need only a few way points because low curvature means that we can use fewer straight lines to approximate the curve.

A curve with high curvature. This task is done similarly to the case of a circle. However, we need many more way points because high curvature means that we cannot use a few straight lines to approximate the curve.

### 7.4 Future Challenges

Future challenges include the following:

**Challenge 2:** Get the robot to fly out of a room from a fixed point in the room.

**Challenge 3:** Get the robot to fly out of a room from any point in the room.

**Challenge 4:** Get the robot to fly out of a room from any point in the room without hitting a static obstacle.

**Challenge 5:** Get the robot to fly out of a room from any point in the room without hitting a dynamic obstacle.
Chapter 8

Future Work

We considered various robotics control problems and techniques in the context of obstacle avoidance. The robot motion planning problem was reduced to a trajectory planning problem through the Minkowski sum. We saw both theoretical and heuristic approaches to optimal control and proposed several geometric ideas to extend those approaches. There are many directions we can go from here:

8.1 Incorporating Geodesics

There is a gap in theory and practice of how to find the geodesics from point $A$ to point $B$ on a manifold in real time. To give an overview, the geodesic is defined as follows. A curve $\gamma(t) = (x_1(t), x_2(t), \ldots, x_n(t))$ is a geodesic if and only if

$$0 = \frac{D}{dt} \left( \frac{d\gamma}{dt} \right),$$

where $\frac{D}{dt}$ represents the covariant derivative, which is the derivative being projected to the tangent plane at a point. The geodesics and their related concepts (such as geodesic flows, geodesic fields) have various minimizing properties (see [do Carmo](#)). This notion definitely has a connection with the optimal trajectory and obstacle avoidance problem that we are concerned with. We can study this connection and flesh out the details.
8.2 Continuing to Integrate Geometry into Quadrotors Obstacle Avoidance Problem

We can extend the “Theories Meet Practices” chapter by carefully developing theories and algorithms to solve the problem. Specifically, we can flesh out on the “Improvement on Curve Matching Algorithm” subsection, and with a new algorithm we can perform an end-to-end test and compare results with the original method.

8.3 Extending Kumar and Belta (2002) Results to Tangent Bundle of SE(3)

Kumar and Belta (2002) presented a method for finding an optimal path in \( \text{SE}(3) \) by lifting up to \( \text{GA}(3) \), the general affine group, and projecting the solution down to get the geodesics in \( \text{SE}(3) \). The general affine group is defined as the semidirect product of the vector space by the general linear group which acts by linear transformations, where the the general linear group \( \text{GL}(3) \) is a group of invertible \( 3 \times 3 \) matrices. From Chapter 4 we can see that this method can be used to generate smooth trajectories for a moving rigid body with specified boundary conditions. However, as we see in Chapter 2 (Configuration of 3D Robot and Quadrotor Model), we can represent the state of the quadrotor by the tangent bundle of \( \text{SE}(3) \). It would be interesting to see whether the projection method from \( \text{GA}(3) \) yields a similar success.

8.4 Tackling More Challenging Obstacle Avoidance Tasks

So far we have only dealt with a known cylinder-shaped obstacle. We can continue to tackle obstacles that have more complex shapes such as arbitrary convex shapes or even non-convex shapes. Currently we also fix the quadrotor to a specific height, reducing the problem to 2.5D, but we can also extend it to the full 3D version where the start point and the goal have different heights.

A much more challenging task would be one that requires a quadrotor to exercise all of its 6 degrees of freedom to avoid an obstacle. We can try to make obstacles become more surprising to the UAV as the algorithms get more advanced and robust. Once algorithms to avoid static obstacles
have been fully developed, we can move on to develop algorithms to avoid dynamic obstacles.

8.5 Simulating before Testing

We have also used a simulator software called X-plane to run a planned trajectory and get data, in addition to the data from real quadrotors.

![Figure 8.1](image.png)

Figure 8.1 A screenshot of a quadrotor model in X-plane. The quadrotor model QRO_X in X-plane is provided by JLN.

There are available open-source files of the quadrotor model. The online quadrotor model can be easily integrated into X-Plane (simply by dragging and dropping files). However, we do not know whether the model is correct (in the sense that it corresponds to the quadrotor model that we have in the real world, i.e. it must have the same dimension, same weight, same rotor speed and control). In other words, the model and the real-world quadrotor are different objects.

In contrast, ROS can record real data from real sensors in the ROSBAG file format. This data as a whole can inter many parameters that should allow us to fine tune the quadrotor model and get a much more accurate representation of the quadrotor. This derived model will have an advan-

1 https://wiki.ros.org/rosbag
tage of fitting the quadrotor that we own (since we collected data from it).
Note that this specificity of the model does not imply the limitation in generalization. The core idea of path planning and algorithms are still the same across platform, but the hardware may be different.

Nevertheless, it is still desirable to have a reliable simulation to test, say, many flight plans without the need to set up the actual drone and refill batteries. We still seek for other simulation software options and also develop our own prototype of flight simulation in MATLAB.

8.6 Studying Quadrotor Swarming

Some other questions to further investigate include, for example, for multiple vehicle teams, how effective swarming behaviors help to complete the desired high-level mission, and how information spreads in the swarm of UAVs. A team of drones can collaboratively work to accomplish a shared mission. However, although we can gain payload by having multiple drones to carry objects, there is a tradeoff with its agile behavior. This consequence is an effect of an increase in the total inertia. Another aspect to consider is how to collectively use drones to gain more information about the environments and how this information propagates among drones. We would again try to use various mathematical tools such as manifolds and Lie algebra. Later on, we can investigate the robustness and stability issues as well.

8.7 Developing an AR.Drone Autonomous Flight iPhone App

The development of an AR.Drone 2.0 control software on a mobile device allows a drone to operate in a much wider range of environments, including outdoor environments. The AR.Drone Open Application Programming Interface (API) Platform provides AR.Drone Software Developer Kit (SDK), which includes source code for an iPhone app called FreeFlight that allows drone communication with an iPhone. Currently, with the FreeFlight app, a user can control the drone with a control pad on iPhone to make the drone translate left, translate right, go up, go down, spin left, spin right, and flip. However, there are no other autonomous flight modes

https://projects.ardrone.org/
to automatically avoid static and dynamic obstacles and achieve high-level
goals. We can add the autonomous button/mode and have users specify
only high-level goals. If successful, we can potentially publish the extended
app in the Apple App Store and ask for feedback to improve our obstacle
avoidance algorithms.

8.8 Integrating Kinect Data to Build a 3D Map; Incorporating Persistent Homology

Microsoft Kinect© is a range sensor device that gives as an output RGB-D
(red-green-blue-depth) point cloud. This data is useful to make a 3D map of
the environment, which can help robots better localize and autonomously
navigate.

The current state-of-the-art technology for this has not considered the
Microsoft Kinect technology to a considerable extent. This technology seems
to perform better than others because of the lower level of noise. And since
the problem of denoising seems to be difficult, a more sophisticated math-
ematical tool must be used. One idea is to look at persistent homology, an
algebraic method for measuring topological nested features of shapes and
functions (see Edelsbrunner and Harer (2008)).
Appendix A

Set-up of Experiments with AR.Drone 2.0

In this section, we provide more in-depth details of our experiment setup and how to set up the system, both hardware and software, in the hope that students continuing this project can successfully set up a working end-to-end framework from the ground up for developing algorithms for quadrotors.

A.1 AR.Drone Specifications

A.1.1 Sensors

- Cameras. There are two cameras: the front camera and the bottom camera
  - 720p 30FPS HD front camera
    - H264 encoding base profile
    - Wide angle lens: 92° diagonal
    - Low latency streaming
    - JPEG photo capture
  - 60 FPS vertical QVGA bottom camera for measuring ground speed
- 3 axis accelerometer ± 50mg precision for measuring all external forces acting upon the quadrotor, including gravity
- Range sensors for determining the distance along a ray
• 3 axis gyroscope 2000°/second precision for measuring orientation
• Ultrasound sensors for measuring ground altitude
• 3 axis magnetometer 6° precision
• Pressure sensor ± 10 Pa precision

The on-board electronics on AR.Drone 2 include 1GHz 32 bit ARM Cortex A8 processor with 800MHz video DSP, TMS320DMC64x, Linux 2.6.32, 1GB DDR2 RAM at 200MHz and Wi-Fi b g n. More specifications about the AR.Drone 2 structure and motors can be found in the official specifications.

A.2 Up and Autonomously Flying AR.Drone

A.2.1 Hardware

• Parrot AR.Drone 2.0
• A laptop, preferably using Ubuntu 12.04 (Precise), with Wi-fi.
• If a laptop does not have Wi-fi, we can use a Wi-fi USB Connector.
• (optional) USB Joystick or control pad (Linux compatible)

A.2.2 Install Robot Operating Systems (ROS)

ROS: Use groovy

Install ROS groovy from Willow Garage. From Dodds’ Robotics Labs Wiki, he has the following remark:

The one instruction I’ve never had to run is the “Configure your Ubuntu repositories” - they always seem OK after a fresh install of Ubuntu

We use the full desktop version, instead of the Virtual Machine, because the desktop version is much faster, which is necessary for many computationally expensive tasks. Be sure to run all of the instructions, including the change to the bashrc script at the bottom of that page. The one at the bottom of the page is easy to miss because the previous step takes a while.

References:
ROS drivers and setup

Set up a /ros_workspace directory and place it in your ROS_PACKAGE_PATH! by running these instructions:

1. Go to your home directory with cd ~
2. Make a directory with mkdir ros_workspace, then go into it
3. It should be the path /home/robot/ros_workspace
4. We need to add this to a file named ~/.bashrc
5. run gedit ~/.bashrc
6. include the line at the bottom:

```
begin{lstlisting}
export ROS_PACKAGE_PATH=/home/robot/ros_workspace:$$
ROS_PACKAGE_PATH
save and close
back at the terminal, run . ~/.bashrc (note the initial dot!)
and then check it with echo \$$ROS_PACKAGE_PATH, which
should show something like /home/robot/ros_workspace:/opt/rosgroovy/stacks
```

Then we install git by running sudo apt-get install git.

ROS new distributions\(^4\) are released approximately once a year, where many bugs in the previous version are fixed, and sometimes the architecture of the operating system has also changed, so the old code will not with the new version. In particular, the groovy version uses a package system called catkin\(^6\) but still supports parts of the old architecture design. When compiling, however, use catkin_make instead of rosmake.

The ROS distribution version we use is called groovy. We can check this through the command rosversion -d. Note that a newer one is called hydro, and an older one is fuerte.

A.2.3 Install the ardrone_autonomy driver

ardrone_autonomy is a ROS driver for Parrot AR.Drone quadrocopter. This driver is based on official AR.Drone SDK version 2.0 and supports both AR.Drone 1.0 and 2.0.

This driver includes many basic functionality to communicate with the drone. It abstracts many low level operations so that developers can focus on high level design. For example, you can directly control the yaw, and the system will figure how to change rotor speed to manipulate the drone to accomplish such an yaw angle. The drivers accept two types of commands, velocity inputs via twist messages\(^5\) and mode changes via empty

\(^{4}\)http://wiki.ros.org/Distributions
\(^{5}\)http://www.ros.org/doc/api/geometry_msgs/html/msg/Twist.html
\(^{6}\)http://wiki.ros.org/catkin
type messages. These commands can be given from either the command line (rostopic echo) or through a compiled node.

The ardrone_autonomy repository can be found here: https://github.com/AutonomyLab/ardrone_autonomy.

The repository is developed by AutonomyLab of Simon Fraser University and other contributions. The README file is very informative. For example, it contains the information about each data parameter that the system logs during AR.Drone flights as follows.

Legacy Navigation Data

Information received from the drone will be published to the ardrone/navdata topic. The message type is ardrone_autonomy::Navdata and contains the following information:

```plaintext
header: ROS message header
batteryPercent: The remaining charge of the drone’s battery (%) 
state: The Drone’s current state:
  * 0: Unknown
  * 1: Initiated
  * 2: Landed
  * 3,7: Flying
  * 4: Hovering
  * 5: Test
  * 6: Taking off
  * 8: Landing
  * 9: Looping
rotX: Left/right tilt in degrees (rotation about the X axis)
rotY: Forward/backward tilt in degrees (rotation about the Y axis)
rotZ: Orientation in degrees (rotation about the Z axis)
magX, magY, magZ: Magnetometer readings (AR-Drone 2.0 Only) (TBA: Convention)
pressure: Pressure sensed by Drone’s barometer (AR-Drone 2.0 Only) (TBA: Unit)
temp: Temperature sensed by Drone’s sensor (AR-Drone 2.0 Only) (TBA: Unit)
wind_speed: Estimated wind speed (AR-Drone 2.0 Only) (TBA: Unit)
wind_angle: Estimated wind angle (AR-Drone 2.0 Only) (TBA: Unit)
winds unsubscribe: Estimated wind angle compensation (AR-Drone 2.0 Only) (TBA: Unit)
altd: Estimated altitude (mm)
motor1..4: Motor PWM values
vx, vy, vz: Linear velocity (mm/s) [TBA: Convention]
ax, ay, az: Linear acceleration (g) [TBA: Convention]
```

http://ros.org/wiki/std_msgs
If you would like to keep track of more data, then take a look at the Selective Navdata (Advanced) section. In addition, you can manually keep track of more data using the publisher and subscriber model. The tutorials can be found in “Up and flying with the AR.Drone and ROS”.

More details about each part of robotics such as how to use the Kinect can be found here.

Sample NAV Data

To get Navdata (navigation data), we run

```bash
$ rostopic echo /ardrone/navdata
```

The navdata message is published by the AR.Drone driver at a rate of 50Hz. Sample data recorded at each time step is as follows:

```plaintext
header:
  seq: 9221
  stamp:
    secs: 1397661748
    nsecs: 111146320
  frame_id: ardrone_base_link
batteryPercent: 37.0
state: 0
magX: 3
magY: 15
magZ: -85
pressure: 96498
temp: 418
wind_speed: 0.0
wind_angle: 0.0
wind_comp_angle: 0.0
rotX: -1.08599996567
rotY: 1.22099995613
rotZ: -18.3159999847
altd: 0
vx: 0.0
vy: -0.0
vz: -0.0
ax: 0.00997698865831
ay: -0.016008593142
az: 0.957161605358
```

In this case, we can also see the tag data, which would help the robot to localize and navigate.

Other important features are also detailed in the README file. In particular, below is how to send commands to AR.Drone: The drone will takeoff, land or emergency stop/reset by publishing an Empty ROS messages to the following topics: ardrone/takeoff, ardrone/land and ardrone/reset respectively.

In order to fly the drone after takeoff, you can publish a message of type geometry_msgs::Twist to the cmd_vel topic. As a reminder of the twist definition in the SE(3) control chapter, twist coordinates, which are in \( \mathbb{R}^6 \), have the following representation:

```
geometry_msgs/Vector3 linear
geometry_msgs/Vector3 angular
```

which translates to the following changes in the states:

```
- linear.x: move backward
+ linear.x: move forward
- linear.y: move right
+ linear.y: move left
- linear.z: move down
+ linear.z: move up
- angular.z: turn left
+ angular.z: turn right
```

The version of the ardrone_autonomy repository that we will be using is a slightly older than the most current one, but it is more stable and well tested. Also, it is modified slightly by the Technical University of Munich (TUM) Computer Vision group in Germany. Below are instructions to install the ardrone_autonomy driver:
A.2.4 Install the tum_ardrone package

The tum_ardrone package builds on the well known monocular SLAM framework Parallel Tracking and Mapping (PTAM), presented by Klein and Murray in their paper at ISMAR07. More information on this part of the software can be found in the original PTAM website[^7] and the corresponding paper[^10]. In addition, the package authors ask us to be aware of the license that comes with it. The tum_ardrone package wiki, which has a link to the source code and instructions to install, can be found here: [http://wiki.ros.org/tum_ardrone](http://wiki.ros.org/tum_ardrone). In this section, we slightly modify the instructions to suit our current setup.

Note that the most up-to-date version (as of April 2014) is changing to the catkin build architecture, but it is yet compatible with the ardrone_autonomy driver. So we need to check out the previous version of this package using the following commands:

```
# cd into ros root dir
roscd

# clone repository

git clone git://github.com/tum-vision/tum_ardrone.git tum_ardrone

# cd into the tum_drone folder
```

[^7]: [http://www.robots.ox.ac.uk/~gk/PTAM/](http://www.robots.ox.ac.uk/~gk/PTAM/)
[^10]: [http://www.robots.ox.ac.uk/~gk/publications.html#2007ISMAR](http://www.robots.ox.ac.uk/~gk/publications.html#2007ISMAR)
cd tum_arдрone

# checkout the appropriate version
git checkout 88d1cc5b139b17c7ba13c7e0d2b7098b657e1748

# add to ros path (if required)
export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:'pwd'/tum_arдрone

# build package (may take up to 10 minutes)
rosmake tum_arдрone

After installing the drivers, we also need to update the ~/.bashrc file so that the next time the computer starts up, we still have the correct path for arдрone_autonomy and tum_arдрone directories.

In ~/.bashrc, add the following lines:

export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:~/catkin_ws/devel/arдрone_autonomy:~/catkin_ws/devel/tum_arдрone

A.2.5 Run

First, we need to connect to the Wi-fi provided by the drone before running these instructions. The Wi-fi from the drone usually has name arдрone_####.

# run roscore
roscore

# in another tab, run driver
rosrun arдрone_autonomy arдрone_driver

# in another tab, run stateestimation node
rosrun tum_arдрone drone_stateestimation

# in another tab, run autopilot node
rosrun tum_arдрone drone_autopilot

# in another tab, run gui node
rosrun tum_arдрone drone_gui

To use joystick, also run the following command in another tab:

rosrun joy joy_node

The graphical user interface (GUI) window allows four modes of control: no control, keyboard control, joystick control, and the autopilot control.
We found that a strategy to develop an autonomy flight algorithm is to have a joystick be able to take over the autopilot during the test.

It is also possible to update the flight planning text file, which resides at the `tum_ardrone/flightPlans` folder. Be sure to set the reference frame to the current position. For example, to make a house-shaped path, use the following path plan:

```
autoInit 500 800

setReference $POSE$
setMaxControl 1
setInitialReachDist 0.2
setStayWithinDist 0.5
setStayTime 0

goto -1 0 -0.4 0

goto 1 0 -0.4 0

goto -1 0 0.8 0

goto 1 0 0.8 0

goto 0 0 1.4 0

goto -1 0 0.8 0

goto -1 0 -0.4 0

goto 1 0 0.8 0

goto 1 0 -0.4 0

goto 0 0 -0.4 0

land
```

### A.2.6 Various tips

- **Run initDemo.txt first.** This will help the drone make the map of environments. Tips from the `tum_ardrone` wiki suggests that you can interrupt the figure anytime by interactively setting a relative target: click on video (relative to current position). However, first fly up at least 1m to facilitate a good scale estimate, do not start by, say, flying horizontally over uneven terrain.

- **Keep track of the battery level.** If the battery is under 20%, the drone will not take off. In the GUI window, you can observe the battery level in the right panel. The current battery that we have runs for about 20 minutes.

- **To help the drone localize, we can help it set up by pointing the front camera to a planar scene.** It is helpful if the scene has many different color points. Then you can see how it tracks these points in the
drone_stateestimation window. Hit space bar. Then, slowly translates the drone to the left or right until there are many matching points shown on the screen. Then hit space bar to let the program compute the point cloud. If the matches are good, then you will see many color points on the screen and the location of the drone with respect to this point cloud.

- If the drone crashes, we need to re-calibrate the drone by taking off the battery and putting it back again.

### A.3 Using Kinect

To install the Kinect drivers on a Linux machine, run the following command: `sudo apt-get install ros-groovy-openni-kinect`. Use the appropriate ROS distribution version. For example, if it is electric, then change the package to `ros-electric-openni-kinect` instead.

Dodds’ Robotics Lab wiki instructions about the Kinect include the following:

```
1 We’ve found that rebooting is sometimes necessary before the Kinect will work...
2 In order to test the Kinect, you’ll need to add a line to the file manifest.xml within the irobot_mudd driver:
3 Go to that folder with roscd irobot_mudd
4 become the superuser with su (you’ll need to type the password )
5 Open and edit manifest.xml and add the line
6 <depend package="cv_bridge"/> directly beneath the line
7 <depend package="nav_msgs"/>
8 Still as superuser, type rosmake -- it’ll take a few seconds to update (our last run took 8.77 seconds)
9 Type exit to stop being superuser and go back to your original username
10 roscore will start ROS in one Terminal tab
11 rosrer openni_camera openni_node in another Terminal tab will start the Kinect drivers. You should see a few INFO and one WARN message...
12 Then, download the Kinect lab’s starter code and save it as kinect_test.py
13 python kinect_test.py in another Terminal tab will start the program... You should see a couple of Gtk warnings
14 The program should open three windows: the RGB image of the scene, a thresholded image, and four sliders
15 (We should have the starter code show the range images, too...)
```
The Kinect lab’s starter code can be found here: https://www.cs.hmc.edu/twiki/bin/view/Robotics/RobotReasoning_Lab4_StarterCode. I have started to learn how to use the Kinect, as shown in Figure A.1
Figure A.1 (top) A Kinect; (bottom) A screenshot of the output of Kinect of Tum waving his hand. The gradient of color shows the depth of objects in the image.
Bibliography


